Table of Contents

《深入理解Node.js：核心思想与源码分析》

Node.js 的源码分析，基于node v6.0.0。

源码分析包括（libuv, v8），需要有一定的 C、C++基础。 Node.js 的源码到处闪烁着开发者的智慧和追求极致的精神。 包括但不限于：

- 系统架构
- 设计模式
- 性能优化
- 奇技淫巧

本书通过分析 node 核心模块的实现，向读者阐述 node 异步 IO，事件循环的核心思想。帮助开发者更好的使用 Node.js。

通过追溯 node 社区开发issue, 探讨 node 的变迁和演进，学习 node.js 的设计哲学。

Table of content

本书版权归作者所有，未经作者授权，禁止一切方式转载。

- 联系作者 @江凌 微博：http://weibo.com/yangjianghua
- 邮箱：yjhjstz#gmail.com， 博客：http://alinode.aliyun.com

本书尚在撰写中，欢迎读者讨论https://github.com/yjhjstz/deep-into-node/issues

如果您觉得还不错, 请我喝杯咖啡，欢迎**Star,** 提交**PR**

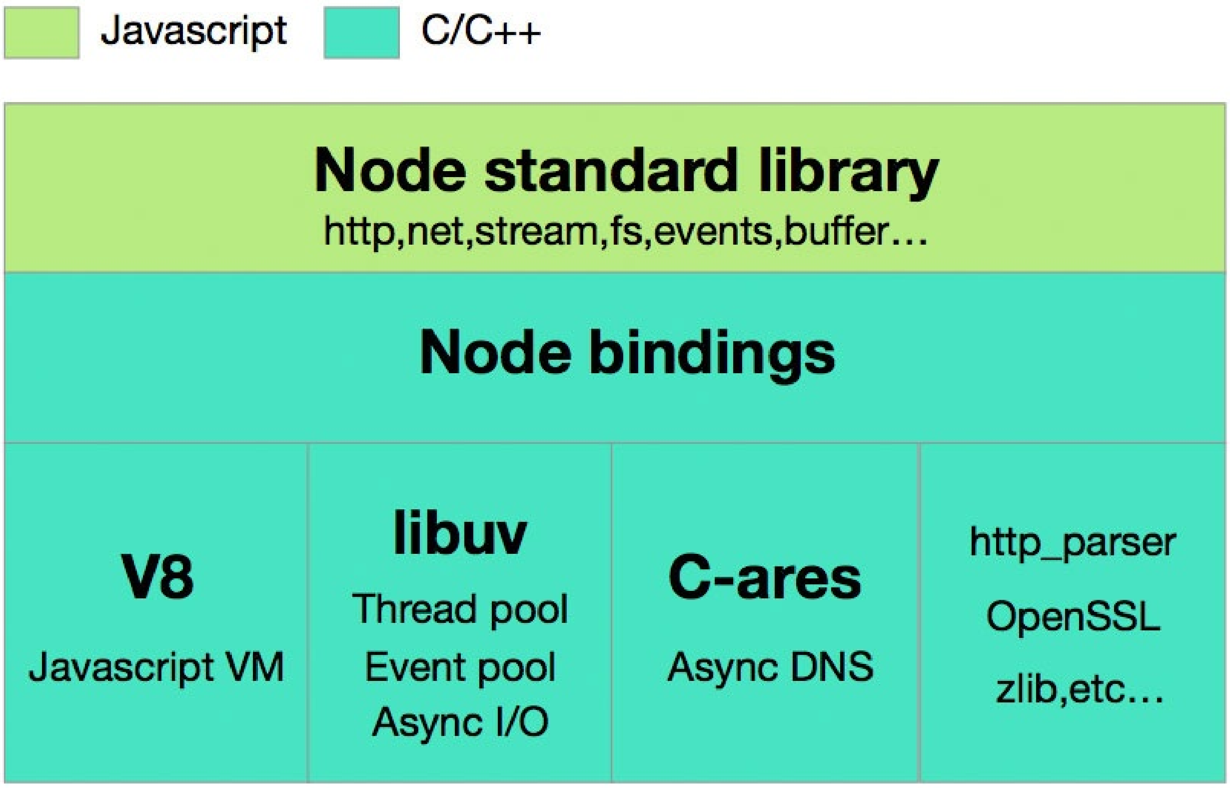
用支付宝扫二维码，加我好友

Chapter1

架构一览

体系架构

Node.js主要分为四大部分，Node Standard Library，Node Bindings，V8，Libuv，架构图如下:



- Node Standard Library 是我们每天都在用的标准库，如Http, Buffer 模块。
- Node Bindings 是沟通JS 和 C++的桥梁，封装V8和Libuv的细节，向上层提供基础API服务。
- 这一层是支撑 Node.js 运行的关键，由 C/C++ 实现。
    - V8 是Google开发的JavaScript引擎，提供JavaScript运行环境，可以说它就是 Node.js 的发动机。
    - Libuv 是专门为Node.js开发的一个封装库，提供跨平台的异步I/O能力.
    - C-ares：提供了异步处理 DNS 相关的能力。
    - http_parser、OpenSSL、zlib 等：提供包括 http 解析、SSL、数据压缩等其他的能力。

代码结构

树形结构查看，使用 tree 命令

```
➜  nodejs git:(master) tree -L 1
.
├── AUTHORS
├── BSDmakefile
├── BUILDING.md
├── CHANGELOG.md
├── CODE_OF_CONDUCT.md
├── COLLABORATOR_GUIDE.md
├── CONTRIBUTING.md
├── GOVERNANCE.md
├── LICENSE
├── Makefile
├── README.md
├── ROADMAP.md
├── WORKING_GROUPS.md
├── android-configure
├── benchmark
├── common.gypi
├── config.gypi
├── config.mk
├── configure
├── deps
├── doc
├── icu_config.gypi
├── lib
├── node.gyp
├── out
├── src
├── test
├── tools
└── vcbuild.bat
```

进一步查看 `deps` 目录：

```
➜  nodejs git:(master) tree deps -L 1
deps
├── cares
├── gtest
├── http_parser
├── npm
├── openssl
├── uv
├── v8
└── zlib
```

`node.js` 核心依赖六个第三方模块。其中核心模块 http_parser, uv, v8这三个模块在后续章节我们会陆续展开。 `gtest` 是C/C++单元测试框架。

为啥是libuv

背景

node.js最初开始于2009年，是一个可以让Javascript代码离开浏览器的执行环境也可以执行的项目。 node.js使用了Google的V8解析引擎和Marc Lehmann的libev。Node.js将事件驱动的I/O模型与适合该模型的编程语言(Javascript)融合在了一起。随着node.js的日益流行，node.js需要同时支持windows, 但是libev只能在Unix环境下运行。Windows 平台上与kqueue(FreeBSD)或者(e)poll(Linux)等内核事件通知相应的机制是IOCP。libuv提供了一个跨平台的抽象，由平台决定使用libev或IOCP。在node-v0.9.0版本中，libuv移除了libev的内容。

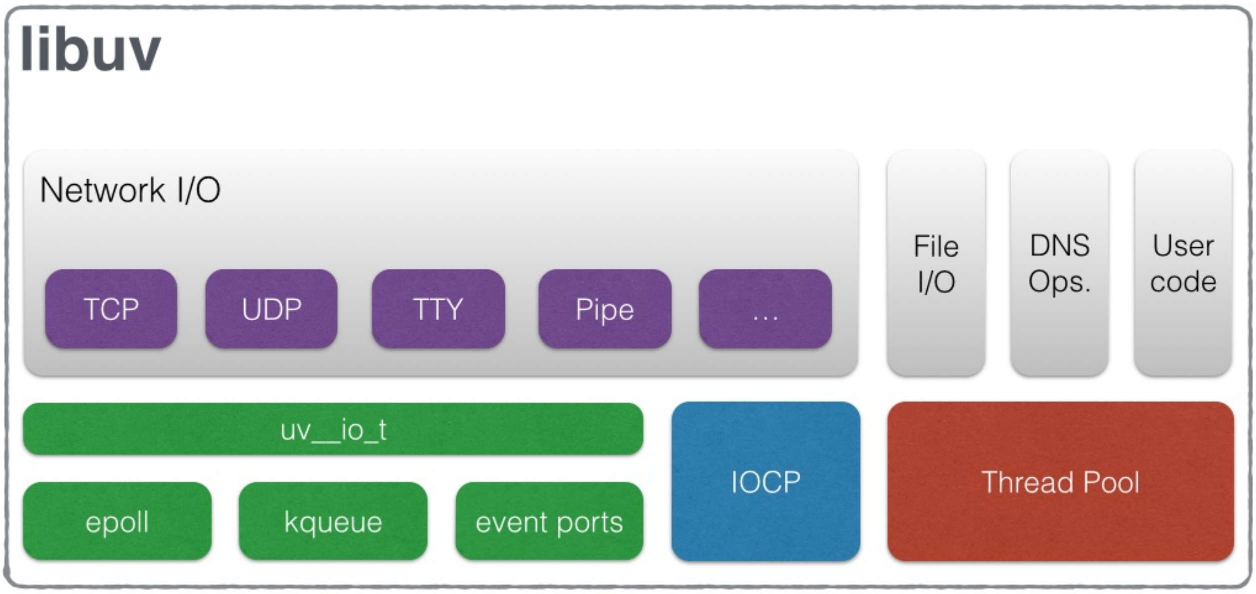
为啥是异步

我们先看一张表：

分类	操作	时间成本
缓存	L1缓存	1纳秒
	L2缓存	4纳秒
	主存储器	100 ns
	SSD 随机读取	16000 ns
I/O	往返在同一数据中心	500000 ns
	物理磁盘寻道	4,000,000 ns

我们看到即便是 SSD 的访问相较于高速的 CPU，仍然是慢速设备。于是基于事件驱动的 IO 模型就应运而生，解决了高速设备同步等待慢速设备或访问的问题。这不是 libuv 的独创，linux kernel 原生支持的 NIO也是这个思路。 但 libuv 统一了网络访问，文件访问，做到了跨平台。

libuv 架构



从左往右分为两部分，一部分是与网络I/O相关的请求，而另外一部分是由文件I/O, DNS Ops以及User code组成的请求。

从图中可以看出，对于Network I/O和以File I/O为代表的另一类请求，异步处理的底层支撑机制是完全不一样的。

对于Network I/O相关的请求， 根据OS平台不同，分别使用Linux上的epoll，OSX和BSD类OS上的kqueue，SunOS上的event ports以及Windows上的IOCP机制。

而对于File I/O为代表的请求，则使用thread pool。利用thread pool的方式实现异步请求处理，在各类OS上都能获得很好的支持。

笔者曾经给 libuv 社区提出过linux 平台下用原生的NIO替换 thread pool 的建议并实现[2],测试发现有3%的提升. 考虑到 NIO 对内核版本的依赖，利用thread pool的方式实现异步请求处理，在各类OS上都能获得很好的支持，相信是 libuv 作者权衡再三的结果。

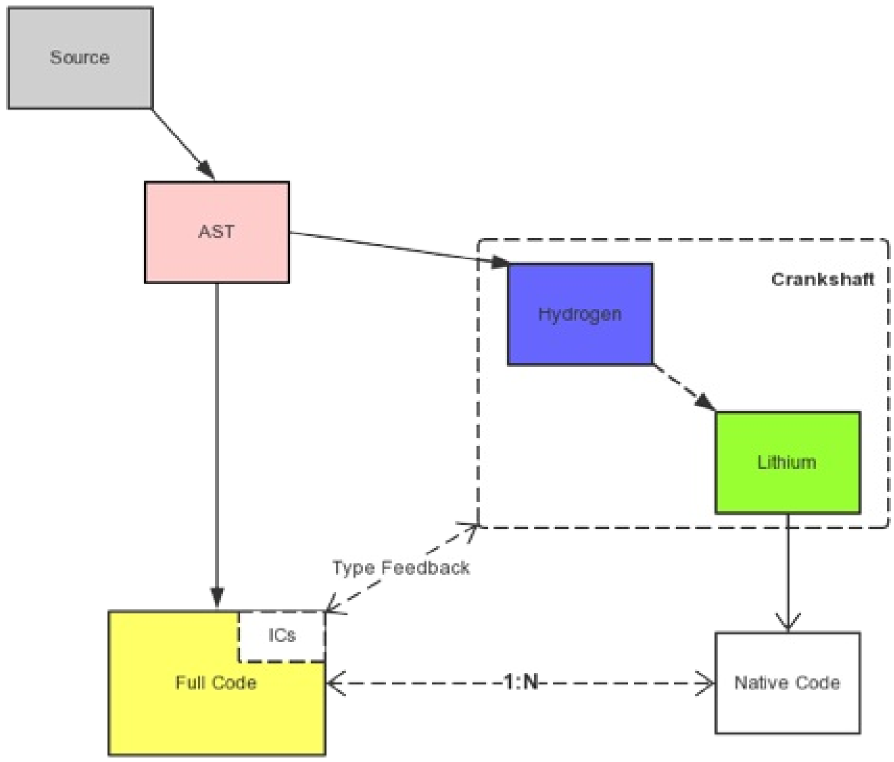
后面详细的模块源码分析时，陆续的会一一剖析。

参考

- [1]. http://luohaha.github.io/Chinese-uvbook/
- [2]. https://github.com/libuv/libuv/issues/461

V8 concept

架构图



现在 JS 引擎的执行过程大致是：源代码 --->抽象语法树 --->字节码 --->JIT--->本地代码。

V8 更加直接的将抽象语法树通过 JIT 技术转换成本地代码，放弃了在字节码阶段可以进行的一些性能优化，但保证了执行速度。 在 V8 生成本地代码后，也会通过 Profiler 采集一些信息，来优化本地代码。虽然，少了生成字节码这一阶段的性能优化， 但极大减少了转换时间。

> PS： Tuborfan 将逐步取代 Crankshaft

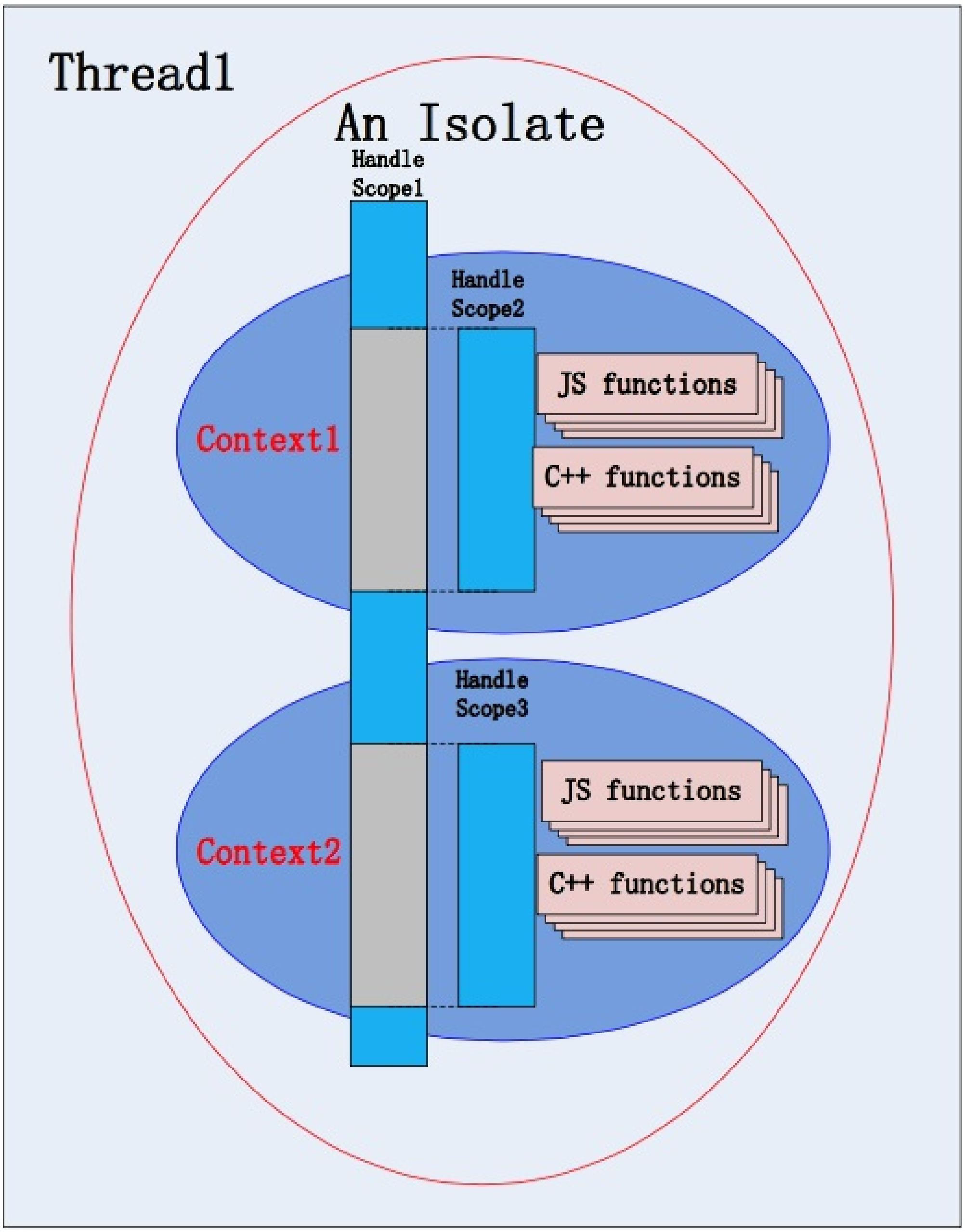
在使用 v8 引擎之前，先来了解一下几个基本概念：句柄（handle），作用域（scope），上下文环境（可以简单地理解为运行环境）。

Isolate

> An isolate is a VM instance with its own heap. It represents an isolated instance of the V8 engine. V8 isolates have completely separate states. Objects from one isolate must not be used in other isolates.

一个 Isolate 是一个独立的虚拟机。对应一个或多个线程。但同一时刻 只能被一个线程进入。所有的 Isolate 彼此之间是完全隔离的，它们不能够有任何共享的资源。如果不显示创建 Isolate，会自动创建一个默认的 Isolate。

后面提到的 Context、Scope、Handle 的概念都是一个 Isolate 内部的，如下图：



Handle 概念

在 V8 中，内存分配都是在 V8 的 Heap 中进行分配的，JavaScript 的值和对象也都存放在 V8 的 Heap 中。这个 Heap 由 V8 独立的去维护，失去引 用的对象将会被 V8 的 GC 掉并可以重新分配给其他对象。而 Handle 即是对 Heap 中对象的引用。V8 为了对内存分配进行管理，GC 需要对 V8 中的 所有对象进行跟踪，而对象都是用 Handle 方式引用的，所以 GC 需要对 Handle 进行管理，这样 GC 就能知道 Heap 中一个对象的引用情况，当一个对象的 Handle 引用发生改变的时候，GC 即可对该对象进行回收或者移动。因此，V8 编程中必须使用 Handle 去引用一个对象，而不是直接通过 C ++ 的方式去获取对象的引用，直接通过 C++ 的方式去引用一个对象，会使得该对象无法被 V8 管理。

Handle 分为 Local 和 Persistent 两种。

从字面上就能知道，Local 是局部的，它同时被 HandleScope 进行管理。persistent，类似与全局的，不受 HandleScope 的管理，其作用域可以延伸到不同的函数，而 Local 是局部的，作用域比较小。 Persistent Handle 对象需要 Persistent::New, Persistent::Dispose 配对使用，类似于 C++ 中 new 和 delete。

Persistent::MakeWeak 可以用来弱化一个 Persistent Handle，如果一个对象的唯一引用 Handle 是一个 Persistent，则可以使用 MakeWeak 方法来弱化该引用，该方法可以触发 GC 对被引用对象的回收。

Scope

从概念上理解，作用域可以看成是一个句柄的容器，在一个作用域里面可以有很多很多个句柄（也就是说，一个 scope 里面可以包含很多很多个 v8 引擎相关的对象），句柄指向的对象是可以一个一个单独地释放的，但是很多时候（真正开始写业务代码的时候），一个一个地释放句柄过于 繁琐，取而代之的是，可以释放一个 scope，那么包含在这个 scope 中的所有 handle 就都会被统一释放掉了。

Scope 在 v8.h 中有这么几个：HandleScope，Context::Scope。

HandleScope 是用来管理 Handle 的，而 Context::Scope 仅仅用来管理 Context 对象。

代码像下面这样：

```
// 在此函数中的 Handle 都会被 handleScope 管理
HandleScope handleScope;
// 创建一个 js 执行环境 Context
Handle<Context> context = Context::New();
Context::Scope contextScope(context);
// 其它代码
```

一般情况下，函数的开始部分都放一个 HandleScope，这样此函数中的 Handle 就不需要再理会释放资源了。 而 Context::Scope 仅仅做了：在构造中调用 context->Enter()，而在析构函数中调用 context->Leave()。

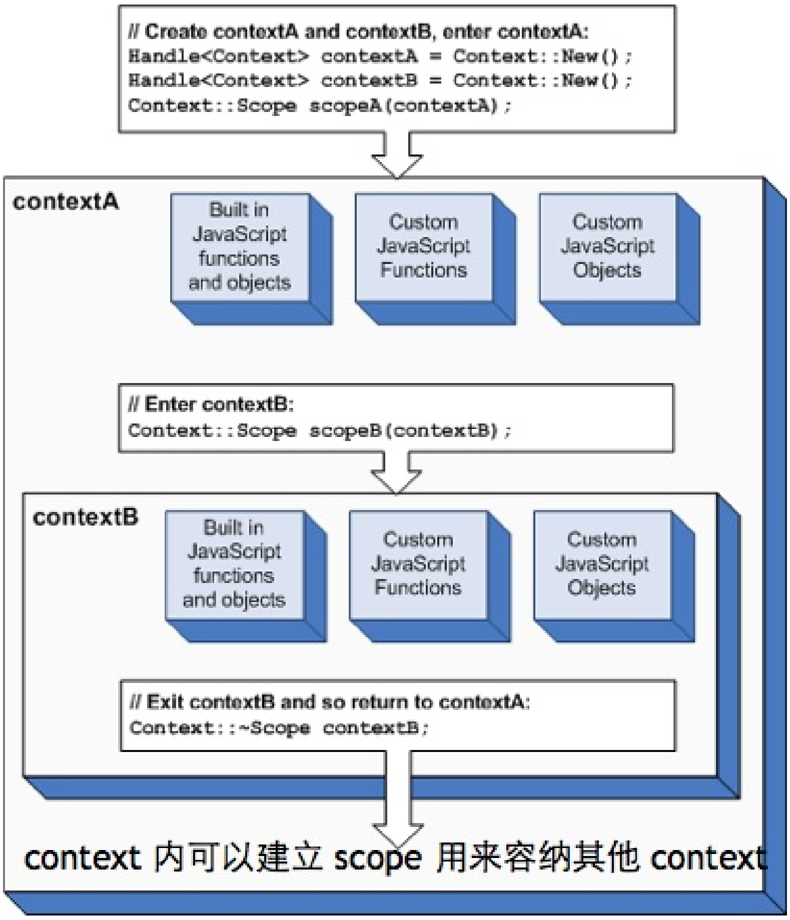
Context 概念

从概念上讲，这个上下文环境也可以理解为运行环境。在执行 javascript 脚本的时候，总要有一些环境变量或者全局函数。 我们如果要在自己的 c++ 代码中嵌入 v8 引擎，自然希望提供一些 c++ 编写的函数或者模块，让其他用户从脚本中直接调用，这样才会体现出 javascript 的强大。 我们可以用 c++ 编写全局函数或者类，让其他人通过 javascript 进行调用，这样，就无形中扩展了 javascript 的功能。

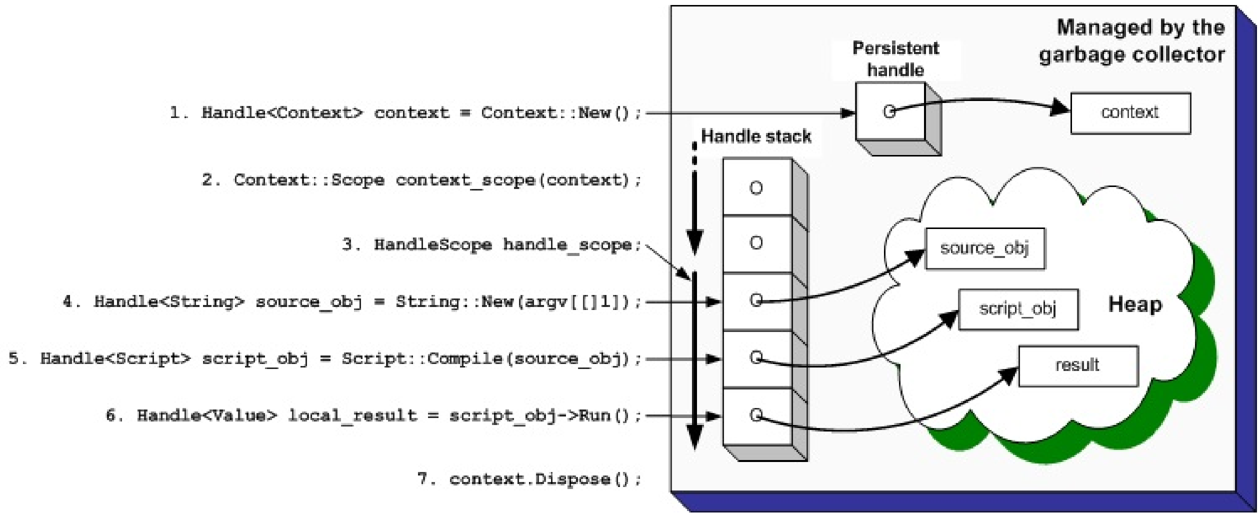
Context 可以嵌套，即当前函数有一个 Context，调用其它函数时如果又有一个 Context，则在被调用的函数中 javascript 是以最近的 Context 为准的，当退出这个函数时，又恢复到了原来的 Context。

我们可以往不同的 Context 里“导入”不同的全局变量及函数，互不影响。据说设计 Context 的最初目的是为了让浏览器在解析 HTML 的 iframe 时，让每个 iframe 都有独立的 javascript 执行环境，即一个 iframe 对应一个 Context。

同作用域下不同的执行上下文



关系



从这张图可以比较清楚的看到 Handle，HandleScope，以及被 Handle 引用的对象之间的关系。从图中可以看到，V8 的对象都是存在 V8 的 Heap 中，而 Handle 则是对该对象的引用。

垃圾回收

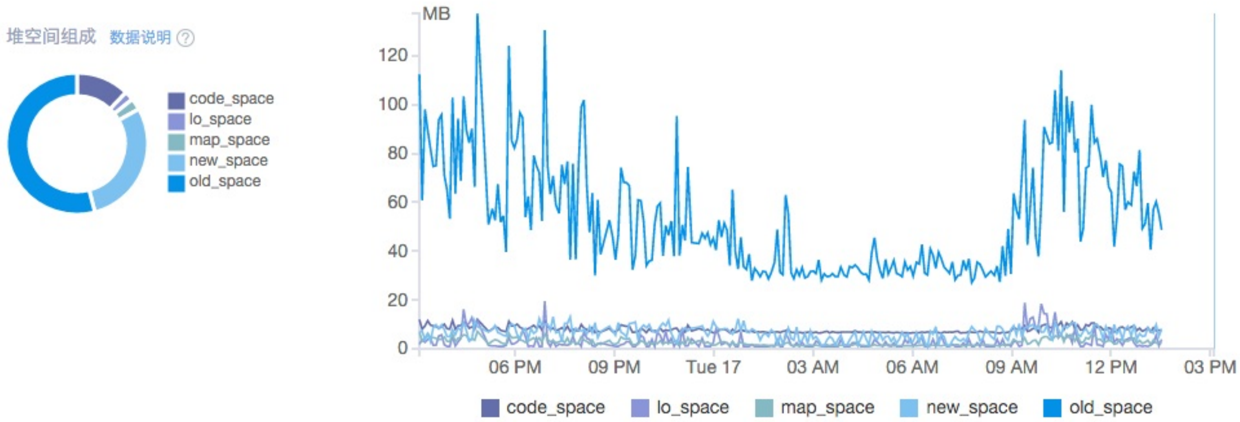
垃圾回收器是一把十足的双刃剑。好处是简化程序的内存管理，内存管理无需程序员来操作，由此也减少了长时间运转的程序的内存泄漏。然而无法预期的停顿，影响了交互体验。

基本概念

垃圾回收器解决基本问题就是，识别需要回收的内存。一旦辨别完毕，这些内存区域即可在未来的分配中重用，或者是返还给操作系统。一个对象当它不是处于活跃状态的时候它就死了。一个对象处于活跃状态，当且仅当它被一个根对象或另一个活跃对象指向。根对象被定义为处于活跃状态，是浏览器或 V8 所引用的对象。比如说全局对象属于根对象，因为它们始终可被访问；浏览器对象，如 DOM 元素，也属于根对象，尽管在某些场合下它们只是弱引用。

堆的构成

在深入研究垃圾回收器的内部工作原理之前，首先来看看堆是如何组织的。V8 将堆分为了几个不同的区域：



新生区 ：大多数对象开始时被分配在这里。新生区是一个很小的区域，垃圾回收在这个区域非常频繁，与其他区域相独立。

老生指针区 ：包含大多数可能存在指向其他对象的指针的对象。大多数在新生区存活一段时间之后的对象都会被挪到这里。

老生数据区 ：这里存放只包含原始数据的对象（这些对象没有指向其他对象的指针）。字符串、封箱的数字以及未封箱的双精度数字数组，在新生区经历一次 Scavenge 后会被移动到这里。

大对象区 ：这里存放体积超过 1MB 大小的对象。每个对象有自己 mmap 产生的内存。垃圾回收器从不移动大对象。

Code 区 ：代码对象，也就是包含 JIT 之后指令的对象，会被分配到这里。

Cell 区、属性 **Cell** 区、**Map** 区 ：这些区域存放 Cell、属性 Cell 和 Map，每个区域因为都是存放相同大小的元素，因此内存结构很简单。

> 如上图：在 node-v4.x 之后，区域进行了合并为：新生区，老生区，大对象区，Map 区，Code 区

有了这些背景知识，我们可以来深入垃圾回收器了。

识别指针

垃圾回收器面临的第一个问题是，如何才能在堆中区分指针和数据，因为指针指向着活跃的对象。大多数垃圾回收算法会将对象在内存中挪动（以便减少内存碎片，使内存紧凑），因此即使不区分指针和数据，我们也常常需要对指针进行改写。
V8 采用了标记指针法：这种方法需要在每个指针的末位预留一位来标记这个字代表的是指针或数据。

对象的晋升

当一个对象经过多次新生代的清理依旧幸存，这说明它的生存周期较长，也就会被移动到老生代，这称为对象的晋升。具体移动的标准有两种：

- 对象从 From 空间复制到 To 空间时，会检查它的内存地址来判断这个对象是否已经活过一次新生代的清理，如果是，则复制到老生代中，否则复制到 To 空间中
- 对象从 From 空间复制到 To 空间时，如果 To 空间已经被使用了超过 25%，那么这个对象直接被复制到老生代。

写屏障

如果新生区中某个对象，只有一个指向它的指针，而这个指针恰好是在老生区的对象当中，我们如何才能知道新生区中那个对象是活跃的呢？ 为了解决这个问题，实际上在写缓冲区中有一个列表 `store-buffer{.cc,.h,-inl.h}` ，列表中记录了所有老生区对象指向新生区的情况。新对象诞生的时候，并不会有指向它的指针，而当有老生区中的对象出现指向新生区对象的指针时，我们便记录下来这样的跨区指向。由于这种记录行为总是发生在写操作时，它被称为写屏障.

垃圾回收三部曲

`Stop-the-World` 的 GC 包括三个主要步骤:

1. 枚举根节点引用;
2. 发现并标记活对象;
3. 垃圾内存清理

分代回收在 V8 中分为 `Scavenge` , `Mark-Sweep` 。

- `Scavenge` : 当分配指针达到了新生区的末尾，就会有一次清理。
- `Mark-Sweep` : 对于活跃超过 2 个小周期的对象，则需将其移动至老生区, 当老生区有足够多的对象时才会触发。

总结

如果你还想了解更多垃圾回收上的东西，我建议你读读 Richard Jones 和 Rafael Lins 写的《Garbage Collection》，这是一个绝好的参考，涵盖了大量你需要了解的内容。你可能还对《Garbage First Garbage-Collection》感兴趣，这是一篇描述 JVM 所使用的垃圾回收算法的论文。

参考

- https://developers.google.com/v8/get_started
- https://developers.google.com/v8/embed
- http://newhtml.net/v8-garbage-collection/

C++ 和 JS 交互

本章主要来讲讲如何通过 V8 来实现 JS 调用 C++。JS 调用 C++，分为 JS 调用 C++ 函数（全局），和调用 C++ 类。

数据及模板

由于 C++ 原生数据类型与 JavaScript 中数据类型有很大差异，因此 V8 提供了 Value 类，从 JavaScript 到 C++，从 C++ 到 JavaScrpt 都会用到这个类及其子类，比如：

```
Handle<Value> Add(const Arguments& args){
   int a = args[0]->Uint32Value();
   int b = args[1]->Uint32Value();

   return Integer::New(a+b);
 }
```

Integer 即为 Value 的一个子类。

V8 中，有两个模板 (Template) 类 (并非 C++ 中的模板类)：

- 对象模板 (ObjectTemplate)
- 函数模板 (FunctionTemplate) 这两个模板类用以定义 JavaScript 对象和 JavaScript 函数。我们在后续的小节部分将会接触到模板类的实例。通过使用 ObjectTemplate，可以将 C++ 中的对象暴露给脚本环境，类似的，FunctionTemplate 用以将 C++ 函数暴露给脚本环境，以供脚本使用。

JS 使用 C++ 变量

在 JavaScript 与 V8 间共享变量事实上是非常容易的，基本模板如下：

```
static char sname[512] = {0};

 static Handle<Value> NameGetter(Local<String> name, const Acces
sorInfo& info) {
     return String::New((char*)&sname,strlen((char*)&sname));
 }

 static void NameSetter(Local<String> name, Local<Value> value,
const AccessorInfo& info) {
   Local<String> str = value->ToString();
   str->WriteAscii((char*)&sname);
 }
```

定义了 NameGetter, NameSetter 之后，在 main 函数中，将其注册在 global 上：

```
 // Create a template for the global object.
 Handle<ObjectTemplate> global = ObjectTemplate::New();
 //public the name variable to script
 global->SetAccessor(String::New("name"), NameGetter, NameSetter
);
```

JS 调用 C++ 函数

在 JavaScript 中调用 C++ 函数是脚本化最常见的方式，通过使用 C++ 函数，可以极大程度的增强 JavaScript 脚本的能力，如文件读写，网络 / 数据库访问，图形 / 图像处理等等，类似于 JAVA 的 jni 技术。

在 C++ 代码中，定义以下原型的函数：

```
 Handle<Value> func(const Arguments& args){//return something}
```

然后，再将其公开给脚本： `global->Set(String::New("func"),FunctionTemplate::New(func));`

JS 使用 C++ 类

如果从面向对象的视角来分析，最合理的方式是将 C++ 类公开给 JavaScript，这样可以将 JavaScript 内置的对象数量大大增加，从而尽可能少的使用宿主语言，而更大的利用动态语言的灵活性和扩展性。事实上，C++ 语言概念众多，内容繁复，学习曲线较 JavaScript 远为陡峭。最好的应用场景是：既有脚本语言的灵活性， 又有 C/C++ 等系统语言的效率。使用 V8 引擎，可以很方便的将 C++ 类” 包装” 成可供 JavaScript 使用的资源。

我们这里举一个较为简单的例子，定义一个 Person 类，然后将这个类包装并暴露给 JavaScript 脚本，在脚本中新建 Person 类的对象，使用 Person 对象的方法。首先，我们在 C++ 中定义好类 Person：

```
class Person {
 private:
   unsigned int age;
   char name[512];

 public:
   Person(unsigned int age, char *name) {
     this->age = age;
     strncpy(this->name, name, sizeof(this->name));
   }

   unsigned int getAge() {
     return this->age;
   }

   void setAge(unsigned int nage) {
     this->age = nage;
   }

   char *getName() {
     return this->name;
   }

   void setName(char *nname) {
     strncpy(this->name, nname, sizeof(this->name));
   }
 };
```

Person 类的结构很简单，只包含两个字段 age 和 name，并定义了各自的 getter/setter. 然后我们来定义构造器的包装：

```
Handle<Value> PersonConstructor(const Arguments& args){
   Handle<Object> object = args.This();
   HandleScope handle_scope;
   int age = args[0]->Uint32Value();

   String::Utf8Value str(args[1]);
   char* name = ToCString(str);

   Person *person = new Person(age, name);
   object->SetInternalField(0, External::New(person));
   return object;
 }
```

从函数原型上可以看出，构造器的包装与上一小节中，函数的包装是一致的，因为构造函数在 V8 看来，也是一个函数。需要注意的是， 从 args 中获取参数并转换为合适的类型之后，我们根据此参数来调用 Person 类实际的构造函数，并将其设置在 object 的内部字段中。紧接着，我们需要包装 Person 类的 getter/setter：

```
 Handle<Value> PersonGetAge(const Arguments& args){
   Local<Object> self = args.Holder();
   Local<External> wrap = Local<External>::Cast(self->GetInterna
lField(0));

   void *ptr = wrap->Value();

   return Integer::New(static_cast<Person*>(ptr)->getAge());
 }

 Handle<Value> PersonSetAge(const Arguments& args) {
   Local<Object> self = args.Holder();
   Local<External> wrap = Local<External>::Cast(self->GetInterna
lField(0));

   void* ptr = wrap->Value();

   static_cast<Person*>(ptr)->setAge(args[0]->Uint32Value());
   return Undefined();
 }
```

而 getName 和 setName 的与上例类似。在对函数包装完成之后，需要将 Person 类暴露给脚本环境： 首先，创建一个新的函数模板，将其与字符串"Person" 绑定，并放入 global：

```
 Handle<FunctionTemplate> person_template = FunctionTemplate::Ne
w(PersonConstructor);
 person_template->SetClassName(String::New("Person"));
 global->Set(String::New("Person"), person_template);
```

然后定义原型模板：

```
 Handle<ObjectTemplate> person_proto = person_template->Prototyp
eTemplate();

 person_proto->Set("getAge", FunctionTemplate::New(PersonGetAge)
);
 person_proto->Set("setAge", FunctionTemplate::New(PersonSetAge)
);

 person_proto->Set("getName", FunctionTemplate::New(PersonGetNam
e));
 person_proto->Set("setName", FunctionTemplate::New(PersonSetNam
e));
```

最后设置实例模板：

```
 Handle<ObjectTemplate> person_inst = person_template->InstanceT
emplate();
 person_inst->SetInternalFieldCount(1);
```

C++ 调用 JS 函数

我们直接看下 src/timer_wrap.cc 的例子，V8 编译执行 timer.js, 构造了 Timer 对象。

```
static void OnTimeout(uv_timer_t* handle) {
    TimerWrap* wrap = static_cast<TimerWrap*>(handle->data);
    Environment* env = wrap->env();
    HandleScope handle_scope(env->isolate());
    Context::Scope context_scope(env->context());
    wrap->MakeCallback(kOnTimeout, 0, nullptr);
}

inline v8::Local<v8::Value> AsyncWrap::MakeCallback(uint32_t ind
ex, int argc, v8::Local<v8::Value>* argv) {
    v8::Local<v8::Value> cb_v = object()->Get(index);
    CHECK(cb_v->IsFunction());
    return MakeCallback(cb_v.As<v8::Function>(), argc, argv);
}
```

`TimerWrap` 对象通过数组的索引寻址，找到 Timer 对象索引 0 的对象，而对其赋值的是在 lib/timer.js 里面的 `list._timer[kOnTimeout] = listOnTimeout;` 。这边找到的对象是个 `Function` , 后面忽略 domains 异常处理等，就是简单的调用 Function 对象的 Call 方法, 并且传人上文提到的 Context 和参数。

`Local<Value> ret = callback->Call(recv, argc, argv);`

这就实现了 C++ 对 JS 函数的调用。

总结

参考

- [1]. https://www.ibm.com/developerworks/cn/opensource/os-cn-v8engine/
- [2]. https://developers.google.com/v8/embed

从「hello world」讲起

先贴一段代码，再熟悉不过，https://nodejs.org/en/about/， 和学习每一种语言一样，从一个简单「hello world」程序对 node.js 有个感性的认识。

```
const http = require('http');
const hostname = '127.0.0.1';
const port = 1337;

http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World\n');
}).listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

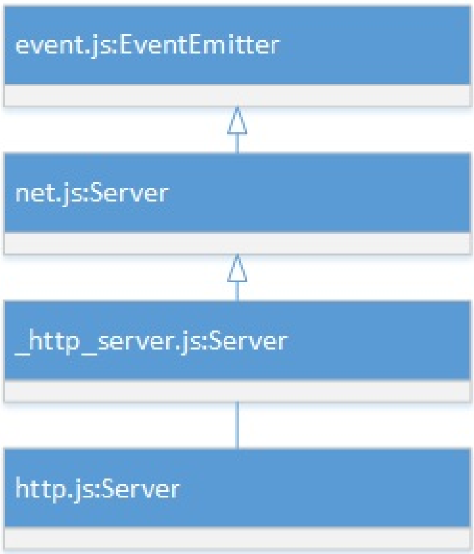
我们从第一句代码看看到底涉及了多少核心模块，让我们开启 node.js 源码分析之旅吧。

第一句： `const http = require('http');` 就涉及到2个模块，分别是module 和 http 模块。

主体代码

```
http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World\n');
}).listen(port, hostname, () => {
  ...
});
```

- 首先了解一下HTTP Server的继承关系，有利于更好的理解代码。



这就又涉及了 event和net模块。

最后一句：

```
console.log(`Server running at http://${hostname}:${port}/`);
```

这里用到了 console模块，但却没有通过 `require` 获取，这就要说到global对象了，Node.js的顶层对象。这里笔者先卖个关子，后面会在 global 章节中详细讲述。

如果想查看 node 的一些调试日志，可以通过设置 NODE_DEBUG 环境变量，比如：

```
NODE_DEBUG=HTTP,STREAM,MODULE,NET node http.js
```

查看 V8的日志

```
node --trace http.js
```

总结

一个简单的 hello world 程序却涉及了多个模块，背后却是 Node社区智慧的结晶，在 web服务，异步 IO 上的高度抽象。真所谓大道至简！

下面以Linus Torvalds的一句名言来开启Node.js的源码之旅吧。

> Talk is cheap, show me the code.

模块

> If V8 is the engine of Node.js, npm is its soul!

npm 世界最大的模块仓库，我们看几个数据：

- ~21 万模块数量
- 每天亿级模块下载量
- 每周 10 亿级的模块下周量

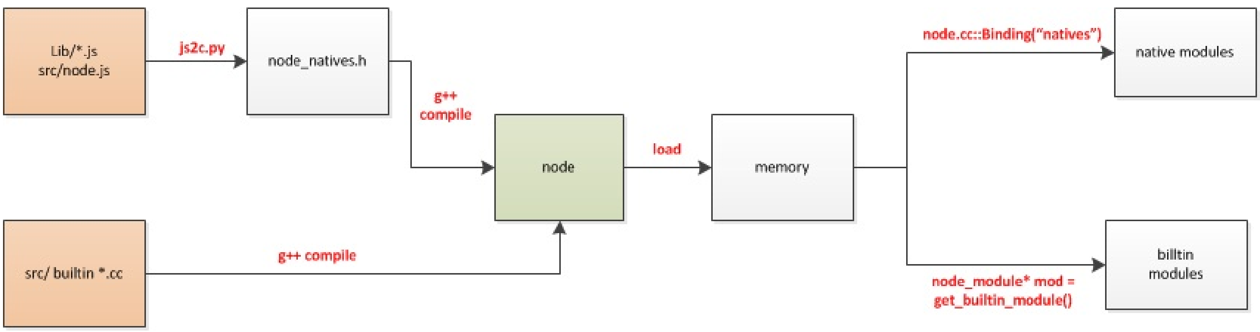
由此诞生了一家做 npm 包管理的公司 `npmjs.com` .

模块加载准备操作

严格来讲，Node 里面分以下几种模块:

- builtin module: Node 中以 c++ 形式提供的模块，如 tcp_wrap、contextify 等
- constants module: Node 中定义常量的模块，用来导出如 signal, openssl 库、文件访问权限等常量的定义。如文件访问权限中的 O_RDONLY，O_CREAT、signal 中的 SIGHUP，SIGINT 等。
- native module: Node 中以 JavaScript 形式提供的模块，如 http,https,fs 等。有些 native module 需要借助于 builtin module 实现背后的功能。如对于 native 模块 buffer , 还是需要借助 builtin node_buffer.cc 中提供的功能来实现大容量内存申请和管理，目的是能够脱离 V8 内存大小使用限制。
- 3rd-party module: 以上模块可以统称 Node 内建模块，除此之外为第三方模块，典型的如 express 模块。

builtin module 和 native module 生成过程



native JS module 的生成过程相对复杂一点，把 node 的源代码下载下来，自己编译后，会在 out/Release/obj/ gen 目录下生成一个文件 `node_natives.h` 。

该文件由 js2c.py 生成。js2c.py 会将 node 源代码中的 lib 目录下所有 js 文件以及 src 目录下的 node.js 文件中每一个字符转换成对应的 ASCII 码，并存放在相应的数组里面。

```
namespace node {
  const char node_native[] = {47, 47, 32, 67, 112 …}

const char console_native[] = {47, 47, 32, 67, 112 …}

const char buffer_native[] = {47, 47, 32, 67, 112 …}

…

}

struct _native {const char name;  const char* source;  size_t so
urce_len;};

static const struct _native natives[] = {{ “node”, node_native,
sizeof(node_native)-1 },

{“dgram”, dgram_native, sizeof(dgram_native)-1 },

{“console”, console_native, sizeof(console_native)-1 },

{“buffer”, buffer_native, sizeof(buffer_native)-1 },

…

}
```

- builtin C++ module 生成过程较为简单。每个 builtin C++ 模块的入口，都会通过宏 NODE_MODULE_CONTEXT_AWARE_BUILTIN 扩展为一个函数。例如对于 tcp_wrap 模块而言，会被扩展为函数 static void _register_tcp_wrap (void) attribute((constructor))。熟悉 GCC 的同学会知道通过 attribute((constructor)) 修饰的函数会在 node 的 main() 函数之前被执行，也就

是说，我们的 builtin C++ 模块会被 main() 函数之前被加载进 modlist_builtin 链表。modlist_builtin 是一个 struct node_module 类型的指针，以它为头，get_builtin_module() 会遍历查找我们需要的模块。

- 对于 node 自身提供的模块，其实无论是 native JS 模块还是 builtin C++ 模块，最终都在编译生成可执行文件时，嵌入到了 ELF 格式的二进制文件 node 里面。

- 而对这两者的提取方式却不一样。对于 JS 模块，使用 process.binding(“natives”)，而对于 C++ 模块则直接用 get_builtin_module() 得到，这部分会在 1.2 节讲述。

module binding

在 node.cc 里面提供了一个函数 Binding()。当我们的应用或者 node 内建的模块调用 require() 来引用另一个模块时，背后的支撑者即是这里提到的 Binding() 函数。后面会讲述这个函数如何支撑 require() 的。这里先主要剖析这个函数。

```
static void Binding(const FunctionCallbackInfo<Value>& args) {
  Environment* env = Environment::GetCurrent(args);

  Local<String> module = args[0]->ToString(env->isolate());
  node::Utf8Value module_v(env->isolate(), module);

  Local<Object> cache = env->binding_cache_object();
  Local<Object> exports;

  if (cache->Has(module)) {
    exports = cache->Get(module)->ToObject(env->isolate());
    args.GetReturnValue().Set(exports);
    return;
  }

  // Append a string to process.moduleLoadList
  char buf[1024];
  snprintf(buf, sizeof(buf), "Binding %s", *module_v);

  Local<Array> modules = env->module_load_list_array();
  uint32_t l = modules->Length();
```

```
  modules->Set(l, OneByteString(env->isolate(), buf));

  node_module* mod = get_builtin_module(*module_v);
  if (mod != nullptr) {
    exports = Object::New(env->isolate());
    // Internal bindings don't have a"module" object, only expor
ts.
    CHECK_EQ(mod->nm_register_func, nullptr);
    CHECK_NE(mod->nm_context_register_func, nullptr);
    Local<Value> unused = Undefined(env->isolate());
    // **for builtin module**
    mod->nm_context_register_func(exports, unused,
      env->context(), mod->nm_priv);
    cache->Set(module, exports);
  } else if (!strcmp(*module_v,"constants")) {
    exports = Object::New(env->isolate());
    // for constants
    DefineConstants(exports);
    cache->Set(module, exports);
  } else if (!strcmp(*module_v,"natives")) {
    exports = Object::New(env->isolate());
    // for native module
    DefineJavaScript(env, exports);
    cache->Set(module, exports);
  } else {
    char errmsg[1024];
    snprintf(errmsg,
             sizeof(errmsg),
             "No such module: %s",
             *module_v);
    return env->ThrowError(errmsg);
  }

  args.GetReturnValue().Set(exports);
}
```

- builtin 优先级最高。对于任何一个需要绑定的模块，都会优先到 builtin 模块列表 modlist_builtin 中去查找。查找过程非常简单，直接遍历这个列表，找到模块名字相同的那个模块即可。找到这个模块后，模块的注册函数会先被执行，

且将一个重要的数据 exports 返回。对于 builtin module 而言，exports object 包含了 builtin C++ 模块暴露出来的接口名以及对于的代码。例如对模块 tcp_wrap 而言，exports 包含的内容可以用如下格式表示：{“TCP”: “/function code of TCPWrap entrance/”, “TCPConnectWrap”: “/function code of TCPConnectWrap entrance/”}。

- constants 模块优先级次之。node 中的常量定义通过 constants 导出。导出的 exports 格式如下: {“SIGHUP”:1, “SIGKILL”:9, “SSL_OP_ALL”: 0x80000BFFL}
- 对于 native module 而言，图 3 中除了数组 node_native 之外，所有的其它模块都会导出到 exports。格式如下：{“_debugger”: _debugger_native , “module”: module_native ，“config”： config_native } 其中，_debugger_native，module_native 等为数组名，或者说就是内存地址。

对比上面三类模块导出的 exports 结构会发现对于每个属性，它们的值代表着完全不同的意义。对于 builtin 模块而言，exports 的 TCP 属性值代表着函数代码入口，对于 constants 模块，SIGHUP 的属性值则代表一个数字，而对于 native 模块，_debugger 的属性值则代表内存地址（准确说应该是 .rodata 段地址）。

模块加载

我们仍旧从 `var http = require('http');` 说起。

require 是怎么来的，为什么平白无故就能用呢，实际上都干了些什么？

- lib/module.js 的中有如下代码。

```
// Loads a module at the given file path. Returns that modul
e's
// `exports` property.
Module.prototype.require = function(path) {
assert(path,'missing path');
assert(typeof path ==='string','path must be a string');
return Module._load(path, this);
};
```

首先 assert 模块进行简单的 path 变量的判断，需要传人的 `path` 是一个 string 类型。

```
// Check the cache for the requested file.
// 1. If a module already exists in the cache: return its export
s object.
// 2. If the module is native: call `NativeModule.require()` wit
h the
//    filename and return the result.
// 3. Otherwise, create a new module for the file and save it to
 the cache.
//    Then have it load  the file contents before returning its
exports
//    object.
Module._load = function(request, parent, isMain) {
  if (parent) {
    debug('Module._load REQUEST %s parent: %s', request, parent.
id);
  }

  var filename = Module._resolveFilename(request, parent);

  var cachedModule = Module._cache[filename];
  if (cachedModule) {
    return cachedModule.exports;
  }

  if (NativeModule.nonInternalExists(filename)) {
    debug('load native module %s', request);
    return NativeModule.require(filename);
  }

  var module = new Module(filename, parent);

  if (isMain) {
    process.mainModule = module;
    module.id = '.';
  }

  Module._cache[filename] = module;

  var hadException = true;
```

```
  try {
    module.load(filename);
    hadException = false;
  } finally {
      if (hadException) {
        delete Module._cache[filename];
      }
  }

  return module.exports;
};
```

- 如果模块在缓存中，返回它的 exports 对象。
- 如果是原生的模块，通过调用 `NativeModule.require()` 返回结果。
- 否则，创建一个新的模块，并保存到缓存中。

让我们再深度遍历的方式查看代码到 `NativeModule.require`.

```
NativeModule.require = function(id) {
  if (id =='native_module') {
    return NativeModule;
  }

  var cached = NativeModule.getCached(id);
  if (cached) {
    return cached.exports;
  }

  if (!NativeModule.exists(id)) {
    throw new Error('No such native module '+ id);
  }

  process.moduleLoadList.push('NativeModule' + id);

  var nativeModule = new NativeModule(id);

  nativeModule.cache();
  nativeModule.compile();

  return nativeModule.exports;
};
```

我们看到，缓存的策略这个贯穿在 node 的实现中。

- 同样的，如果在 cache 中存在，则直接返回 exports 对象。
- 如果不在，则加入到 `moduleLoadList` 数组中，创建新的 NativeModule 对象。

下面是最关键的一句

```
nativeModule.compile();
```

具体实现在 `node.js` 中：

```
NativeModule.getSource = function(id) {
  return NativeModule._source[id];
};

NativeModule.wrap = function(script) {
  return NativeModule.wrapper[0] + script + NativeModule.wrapper[
1];
};

NativeModule.wrapper = ['(function (exports, require, module, __
filename, __dirname) {','\n});' ];

NativeModule.prototype.compile = function() {
  var source = NativeModule.getSource(this.id);
  source = NativeModule.wrap(source);

  var fn = runInThisContext(source, {
    filename: this.filename,
    lineOffset: 0
  });
  fn(this.exports, NativeModule.require, this, this.filename);

  this.loaded = true;
};
```

`wrap` 函数将 http.js 包裹起来, 交由 `runInThisContext` 编译源码，返回 fn 函数, 依次将参数传人。

process

先看看 node.js 的底层 C++ 传递给 javascript 的一个变量 process，在一开始运行 node.js 时，程序会先配置好 process `Handleprocess = SetupProcessObject(argc, argv);`

- 然后把 process 作为参数去调用 js 主程序 src/node.js 返回的函数，这样 process 就传递到 javascript 里了。

```
//node.cc

// 通过 MainSource() 获取已转化的 src/node.js 源码，并执行它

Local f_value = ExecuteString(MainSource(), IMMUTABLE_STRING(“no
de.js”));
// 执行 src/node.js 后获得的是一个函数，从 node.js 源码可以看出：

//node.js

//(function(process) {

//    global = this;

//    …

//})

Local f = Local::Cast(f_value);
// 创建函数执行环境，调用函数，把 process 传入

Localglobal = v8::Context::GetCurrent()->Global();

Local args[1] = {
  Local::New(process)
};

f->Call(global, 1, args);
```

vm

`runInThisContext` 又是怎么一回事呢？

```
  var ContextifyScript = process.binding('contextify').Contextif
yScript;
  function runInThisContext(code, options) {
    var script = new ContextifyScript(code, options);
    return script.runInThisContext();
  }
```

- node.cc 的 Binding 中有如下调用，对模块进行注册， `mod->nm_context_register_func(exports, unused, env->context(), mod->nm_priv);`

我们看下 node.h 中 mod 数据结构的定义：

```
struct node_module {
  int nm_version;
  unsigned int nm_flags;
  void* nm_dso_handle;
  const char* nm_filename;
  node::addon_register_func nm_register_func;
  node::addon_context_register_func nm_context_register_func;
  const char* nm_modname;
  void* nm_priv;
  struct node_module* nm_link;
};
```

- node.h 中还有如下宏定义，接着往下看！

```
#define NODE_MODULE_CONTEXT_AWARE_X(modname, regfunc, priv, flag
s)    \
  extern "C" {
      \
    static node::node_module _module =
      \
    {
      \
      NODE_MODULE_VERSION,
      \
      flags,
```

```
        \
        NULL,
        \
        __FILE__,
        \
        NULL,
        \
        (node::addon_context_register_func) (regfunc),
        \
        NODE_STRINGIFY(modname),
        \
        priv,
        \
        NULL
        \
      };
        \
      NODE_C_CTOR(_register_ ## modname) {
        \
        node_module_register(&_module);
        \
      }
        \
    }

#define NODE_MODULE_CONTEXT_AWARE_BUILTIN(modname, regfunc)
        \
  NODE_MODULE_CONTEXT_AWARE_X(modname, regfunc, NULL, NM_F_BUILT
IN)   \
```

- node_contextify.cc 中有如下宏调用，终于看清楚了！结合前面几点，实际上就是把 node_module 的 nm_context_register_func 与 node::InitContextify 进行了绑定。

```
NODE_MODULE_CONTEXT_AWARE_BUILTIN(contextify, node::InitContexti
fy);
```

我们回溯而上，通过 `node_module_register(&_module);` , `process.binding('contextify')` --> `mod->nm_context_register_func(exports, unused, env->context(), mod->nm_priv);` --> `node::InitContextify()` .

这样通过 `env->SetProtoMethod(script_tmpl,"runInThisContext", RunInThisContext);` ，绑定了『runInThisContext』和 `RunInThisContext` .

runInThisContext 是将被包装后的源字符串转成可执行函数，（runInThisContext 来自 contextify 模块），runInThisContext 的作用，类似 eval，再执行这个被 eval 后的函数。

这样就成功加载了 `native` 模块, 标记 this.loaded = true;

总结

Node.js 通过 cache 解决无限循环引用的问题, 也是系统优化的重要手段，通过以空间换时间，使得每次加载模块变得非常高效。

在实际的业务开发中，我们从堆的角度观察 node 启动模块后，缓存了大量的模块，包括第三方的模块，有的可能只加载使用一次。笔者觉得有必要有一种模块的卸载机制 [1], 可以降低对 V8 堆内存的占用，从而提升后续垃圾回收的效率。

参考

[1].https://github.com/nodejs/node/issues/5895

Global对象

所有属性都可以在程序的任何地方访问，即全局变量。在javascript中，通常window是全局对象，而node.js的全局对象是global，所有全局变量都是global对象的属性，如：console、process等。

全局对象与全局变量

global最根本的作用是作为全局变量的宿主。满足以下条件成为全局变量。

- 在最外层定义的变量
- 全局对象的属性
- 隐式定义的变量（未定义直接赋值的变量）

node.js中不可能在最外层定义变量，因为所有的用户代码都是属于当前模块的，而模块本身不是最外层上下文。node.js中也不提倡自定义全局变量。

Node提供以下几个全局对象，它们是所有模块都可以调用的。

- global：表示Node所在的全局环境，类似于浏览器的window对象。需要注意的是，如果在浏览器中声明一个全局变量，实际上是声明了一个全局对象的属性，比如var x = 1等同于设置window.x = 1，但是Node不是这样，至少在模块中不是这样（REPL环境的行为与浏览器一致）。在模块文件中，声明var x = 1，该变量不是global对象的属性，global.x等于undefined。这是因为模块的全局变量都是该模块私有的，其他模块无法取到。

- process：该对象表示Node所处的当前进程，允许开发者与该进程互动。

- console：指向Node内置的console模块，提供命令行环境中的标准输入、标准输出功能。

Node还提供一些全局函数。

- setTimeout()：用于在指定毫秒之后，运行回调函数。实际的调用间隔，还取决于系统因素。间隔的毫秒数在1毫秒到2,147,483,647毫秒（约24.8天）之间。如果超过这个范围，会被自动改为1毫秒。该方法返回一个整数，代表这个新建定时器的编号。
- clearTimeout()：用于终止一个setTimeout方法新建的定时器。

- setInterval()：用于每隔一定毫秒调用回调函数。由于系统因素，可能无法保证每次调用之间正好间隔指定的毫秒数，但只会多于这个间隔，而不会少于它。指定的毫秒数必须是1到2,147,483,647（大约24.8天）之间的整数，如果超过这个范围，会被自动改为1毫秒。该方法返回一个整数，代表这个新建定时器的编号。
- clearInterval()：终止一个用setInterval方法新建的定时器。
- require()：用于加载模块。
- Buffer()：用于操作二进制数据。

Node提供两个全局变量，都以两个下划线开头。

- _filename：指向当前运行的脚本文件名。
- _dirname：指向当前运行的脚本所在的目录。 除此之外，还有一些对象实际上是模块内部的局部变量，指向的对象根据模块不同而不同，但是所有模块都适用，可以看作是伪全局变量，主要为module, module.exports, exports等。

module.exports vs exports

如果想不借助global，在不同模块之间共享代码，就需要用到exports属性。令人有些迷惑的是，在node.js里，还有另外一个属性，是module.exports。一般情况下，这2个属性的作用是一致的，但是如果对exports或者module.exports赋值的话，又会呈现出令人奇怪的结果。

首先，exports和module.exports都是某个对象的引用（reference），初始情况下，它们指向同一个object，如果不修改module.exports的引用的话，这个object稍后会被导出。

```
exports  module.exports
  |         /
  |        /
  V       V
   Object
```

所以如果只是给对象添加属性，不改变exports和module.exports的引用目标的话，是完全没有问题的。

但是有时候，希望导出的是一个构造函数，那么一般会这么写：

```
// b.js
module.exports = function (name, age) {
    this.name = name;
    this.age = age;
}

exports.sex = "male";
```

```
var Person = require("./b");
var person = new Person("Tony", 33);
console.log(person); // {name:"Tony", age:33}
console.log(Person.sex); // undefined
```

这个sex属性不会导出，因为引用关系已经改变：

```
  exports  module.exports
    |          |
    |          |
    V          V
   function   Object
```

而node.js导出的，永远是module。exports指向的对象，在这里就是function。所以exports指向的那个object，现在已经不会被导出了，为其增加的属性当然也就没用了。

如果希望把sex属性也导出，就需要这样写：

```
exports = module.exports = function (name, age) {
    this.name = name;
    this.age = age;
}

exports.sex = "male";
```

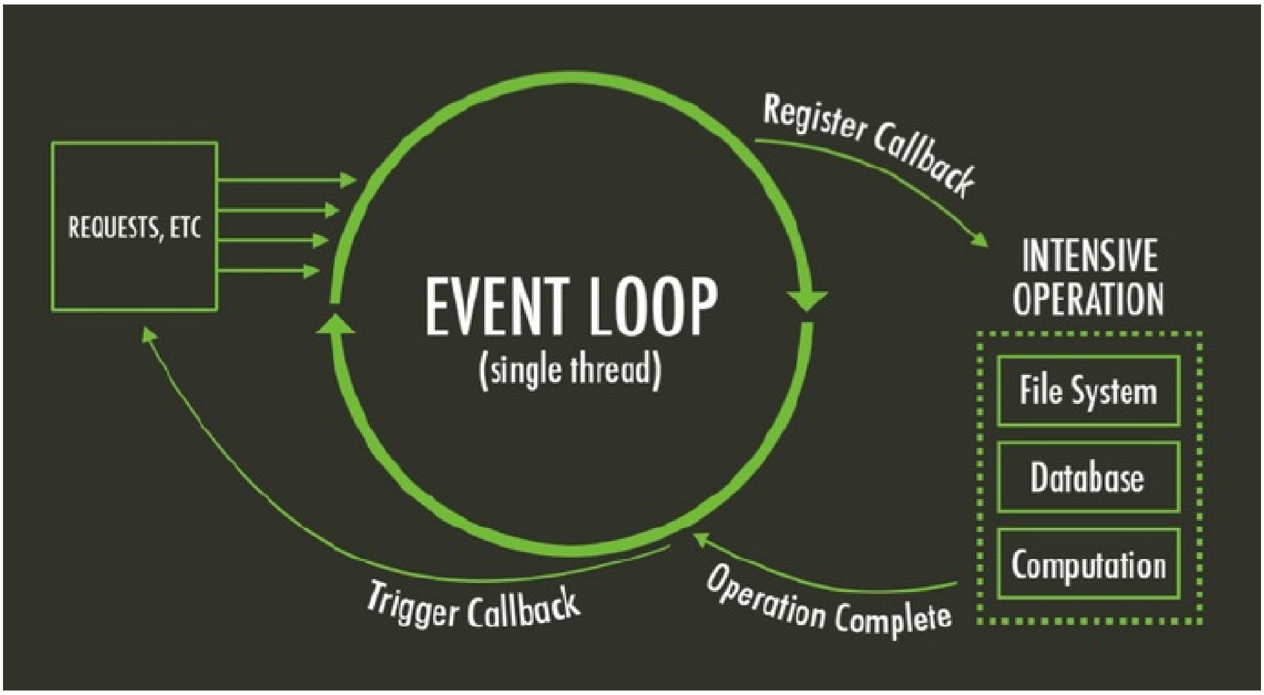
总结

- node.js 设计的2个导出引用的对象，反而增加了迷惑性。
- 避免污染全局空间。

参考

事件循环

> "Event Loop是一个程序结构，用于等待和发送消息和事件。（a programming construct that waits for and dispatches events or messages in a program.）"



事件循环

事件循环的职责，就是不断得等待事件的发生，然后将这个事件的所有处理器，以它们订阅这个事件的时间顺序，依次执行。当这个事件的所有处理器都被执行完毕之后，事件循环就会开始继续等待下一个事件的触发，不断往复。

当同时并发地处理多个请求时，以上的概念也是正确的，可以这样理解：在单个的线程中，事件处理器是一个一个按顺序执行的。

即如果某个事件绑定了两个处理器，那么第二个处理器会在第一个处理器执行完毕后，才开始执行。在这个事件的所有处理器都执行完毕之前，事件循环不会去检查是否有新的事件触发。在单个线程中，一切都是有顺序地一个一个地执行的！

Node.js 中的事件循环

Node采用V8作为JavaScript的执行引擎，同时使用libuv实现事件驱动式异步I/O。其事件循环就是采用了libuv的默认事件循环。 在src/node.cc中，

```
Environment* env = CreateEnvironment(
        node_isolate,
        uv_default_loop(),
        context,
        argc,
        argv,
        exec_argc,
        exec_argv);
```

这段代码建立了一个node执行环境，可以看到第三行的uv_default_loop()，这是libuv库中的一个函数，它会初始化uv库本身以及其中的default_loop_struct，并返回一个指向它的指针default_loop_ptr。 之后，Node会载入执行环境并完成一些设置操作，然后启动event loop：

```
bool more;
do {
  more = uv_run(env->event_loop(), UV_RUN_ONCE);
  if (more == false) {
    EmitBeforeExit(env);
    // Emit `beforeExit` if the loop became alive either after e
mitting
    // event, or after running some callbacks.
    more = uv_loop_alive(env->event_loop());
    if (uv_run(env->event_loop(), UV_RUN_NOWAIT) != 0)
      more = true;
  }
} while (more == true);
code = EmitExit(env);
RunAtExit(env);
```

more用来标识是否进行下一轮循环。 env->event_loop()会返回之前保存在env中的default_loop_ptr，uv_run函数将以指定的UV_RUN_ONCE模式启动libuv的event loop。在这种模式下，uv_run会至少处理一个事件：这意味着，如果当前事件队列中没有需要处理的I/O事件，uv_run会阻塞住，直到有I/O事件需要处理，或者下一个定时器时间到。如果当前没有I/O事件也没有定时器事件，则uv_run返回false。

接下来Node会根据more的情况决定下一步操作：

- 如果more为true，则继续运行下一轮loop。
- 如果more为false，说明已经没有等待处理的事件了，EmitBeforeExit(env);触发进程的'beforeExit'事件，检查并处理相应的处理函数，完成后直接跳出循环。

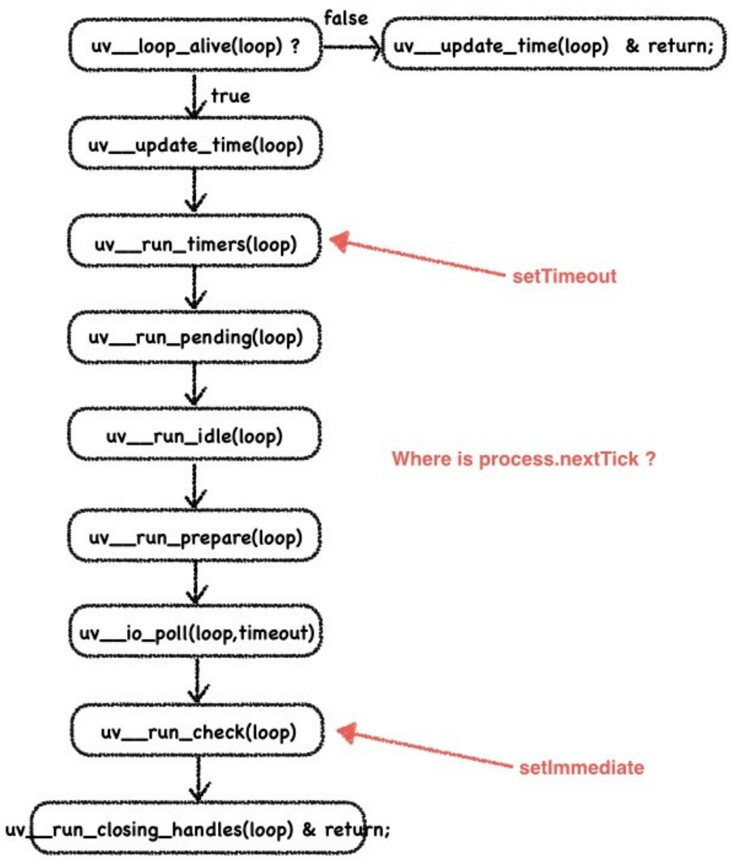
最后触发'exit'事件，执行相应的回调函数，Node运行结束，后面会进行一些资源释放操作。

在libuv中，event loop会在每次循环的开始更新自己的time从而实现计时功能，而I/O事件则分为两类：

- Network I/O是使用系统提供的非阻塞式I/O解决方案，例如在Linux上使用epoll，windows上使用IOCP。
- 文件操作和DNS操作没有（很好的）系统解决方案，因此libuv自建了线程池，在其中进行阻塞式I/O。

另外我们也可以将自定义的函数抛到线程池中运行，在运行结束后主线程会执行相应的回调函数，不过Node并没有将这一项功能加入到JavaScript中，也就是说只用原生Node是无法在JavaScript中开启新的线程进行并行执行的。

process.nextTick



带着这个问题，我们看看 JS 层的 nextTick 是怎么被驱动。

在入口点 `src/node.js`，`processNextTick` 方法构建了 `process.nextTick` API。

`process._tickCallback` 作为 nexttick 的回调函数，挂到了 `process` 对象上，由 C++层面回调使用。

```
startup.processNextTick = function() {
    var nextTickQueue = [];
    var pendingUnhandledRejections = [];
    var microtasksScheduled = false;

    // Used to run V8's micro task queue.
    var _runMicrotasks = {};

    // *Must* match Environment::TickInfo::Fields in src/env.h.
    var kIndex = 0;
    var kLength = 1;

    process.nextTick = nextTick;
    // Needs to be accessible from beyond this scope.
    process._tickCallback = _tickCallback;
    process._tickDomainCallback = _tickDomainCallback;

    // This tickInfo thing is used so that the C++ code in src/n
ode.cc
    // can have easy access to our nextTick state, and avoid unn
ecessary
    // calls into JS land.
    const tickInfo = process._setupNextTick(_tickCallback, _runM
icrotasks);
    // 省略...
}
```

通过 `process._setupNextTick` 注册 `_tickCallback` 到 `env` 的 `tick_callback_function` 上。

在 `src/async_wrap.cc` 文件中，我们发现对其的调用如下：

```
Local<Value> AsyncWrap::MakeCallback(const Local<Function> cb,
                                     int argc,
                                     Local<Value>* argv) {
  // ...
  Environment::TickInfo* tick_info = env()->tick_info();

  if (tick_info->in_tick()) {
    return ret;
  }

  if (tick_info->length() == 0) {
    env()->isolate()->RunMicrotasks();
  }

  if (tick_info->length() == 0) {
    tick_info->set_index(0);
    return ret;
  }

  tick_info->set_in_tick(true);

  env()->tick_callback_function()->Call(process, 0, nullptr);

  tick_info->set_in_tick(false);
  // ...
```

当无 `nextTick` 任务时， `env()->isolate()->RunMicrotasks();` 会驱动 `Promise` 任务执行。

否则会调用 `tick_callback_function` ,也就是 `_tickCallback` 。

看到这里我也有个疑问，如果没有异步 IO 呢，怎么驱动呢？

我们来到 `lib/module.js` , 如下

```
// bootstrap main module.
Module.runMain = function() {
  // Load the main module--the command line argument.
  Module._load(process.argv[1], null, true);
  // Handle any nextTicks added in the first tick of the program
  process._tickCallback();
};
```

`Module._load` 加载主脚本后，就调用 `_tickCallback`，处理第一次的 tick 了。

所以上面的疑问有了答案，`nextTick` 主要在 `uv__io_poll` 驱动。为什么说主要呢? 因为还 可能在 Timer模块驱动，具体细节留给读者去研究啦。

总结

参考

- http://acemood.github.io/2016/02/01/event-loop-in-javascript/

Timer源码解读

涉及源码

- lib/timers.js
- src/timer_wrap.cc
- deps/uv/src/unix/timer.c
- deps/uv/src/heap-inl.h

主要分为 javascript 层面的实现和 `libuv` 层面的实现, 而timer_wrap.cc 作为一个bridge,完成 javascript 和 C++的交互调用。

使用场景

定时器主要的使用场景或者说适用场景：

- 定时任务，比如业务中定时检查状态等；
- 超时控制，比如网络超时控制重传。

在 node.js 的实现中，

```
function responseOnEnd() {
    // 省略
    debug('AGENT socket keep-alive');
    if (req.timeoutCb) {
      socket.setTimeout(0, req.timeoutCb);
      req.timeoutCb = null;
    }
 }
`
```

你可能会有疑问：为啥在 HTTP 模块要用呢？

我们知道HTTP协议采用“请求-应答”模式，当使用普通模式，即非KeepAlive模式时，每个请求/应答客户和服务器都要新建一个连接，完成之后立即断开连接（HTTP协议为无连接的协议）；当使用Keep-Alive模式（又称持久连接、连接重

用）时，Keep-Alive功能使客户端到服务器端的连接持续有效，当出现对服务器的后继请求时，Keep-Alive功能避免了建立或者重新建立连接。

```
  if (req.httpVersionMajor < 1 || req.httpVersionMinor < 1) {
    this.useChunkedEncodingByDefault = chunkExpression.test(req.
headers.te);
    this.shouldKeepAlive = false;
  }
```

HTTP/1.0中默认是关闭的，需要在http头加入"Connection: Keep-Alive"，才能启用Keep-Alive；http/1.1中默认启用Keep-Alive，如果加入"Connection: close"，才关闭。

目前大部分浏览器都是用HTTP/1.1协议，也就是说默认都会发起Keep-Alive的连接请求了，Node.js 针对2种协议按上述代码做了判断处理。

当然了，这个连接不能就这么一直保持着，所以一般都会有一个超时时间，超过这个时间客户端还没有发送新的http请求，那么服务器就需要自动断开从而继续为其他客户端提供服务。 Node.js的HTTP 模块对于每一个新的连接创建一个 socket 对象，调用socket.setTimeout设置一个定时器用于超时后自动断开连接。

数据结构选择

一个Timer本质上是这样的一个数据结构：deadline越近的任务拥有越高优先级，提供以下3种基本操作：

- schedule 新增任务
- cancel 删除任务
- expire 执行到期的任务

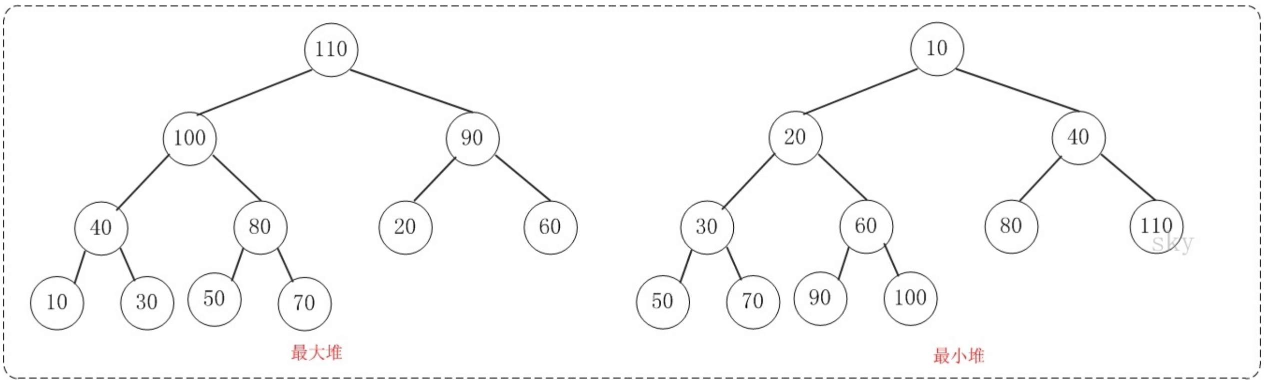
实现方式	**schedule**	**cancel**	**expire**
基于链表	O(1)	O(n)	O(n)
基于排序链表	O(n)	O(1)	O(1)
基于最小堆	O(lgn)	O(1)	O(1)
基于时间轮	O(1)	O(1)	O(1)

timer 的实现历经变迁，每次变迁都是思维碰撞的火花，让我们走进源码，细细品味。

libuv 实现

数据结构-最小堆

最小堆首先是二叉堆，二叉堆是完全二元树或者是近似完全二元树，它分为两种：最大堆和最小堆。 最大堆：父结点的键值总是大于或等于任何一个子节点的键值；最小堆：父结点的键值总是小于或等于任何一个子节点的键值。示意图如下：



节点定义在deps/uv/src/heap-inl.h，如下:

```
struct heap_node {
  struct heap_node* left;
  struct heap_node* right;
  struct heap_node* parent;
};
```

根节点定义：

```
/* A binary min heap.  The usual properties hold: the root is th
e lowest
 * element in the set, the height of the tree is at most log2(no
des) and
 * it's always a complete binary tree.
 *
 * The heap function try hard to detect corrupted tree nodes at
the cost
 * of a minor reduction in performance.  Compile with -DNDEBUG t
o disable.
 */
struct heap {
  struct heap_node* min;
  unsigned int nelts;
};
```

这边我们可以清楚的看到，最小堆采用指针组织数据，而不是数组。 `min` 始终指向最小的节点如果存在的话。作为一个排序的集合，它还需要一个用户指定的比较函数，决定哪个节点更小，或者说当过期时间一样时，决定他们的次序。毕竟没有规则不成方圆。

```
static int timer_less_than(const struct heap_node* ha,
                           const struct heap_node* hb) {
  const uv_timer_t* a;
  const uv_timer_t* b;

  a = container_of(ha, const uv_timer_t, heap_node);
  b = container_of(hb, const uv_timer_t, heap_node);

  if (a->timeout < b->timeout)
    return 1;
  if (b->timeout < a->timeout)
    return 0;

  /* Compare start_id when both have the same timeout. start_id
is
   * allocated with loop->timer_counter in uv_timer_start().
   */
  if (a->start_id < b->start_id)
    return 1;
  if (b->start_id < a->start_id)
    return 0;

  return 0;
}
```

这边我们可以看到，首先比较两者的 `timeout` ，如果二者一样，则比较二者被 `schedule` 的 id, 该 id 由 `loop->timer_counter` 递增生成，在调用 `uv_timer_start` 时赋值给 `start_id` .

具体实现

```
62 int uv_timer_start(uv_timer_t* handle,
63                    uv_timer_cb cb,
64                    uint64_t timeout,
65                    uint64_t repeat) {
66   uint64_t clamped_timeout;
67
68   if (cb == NULL)
69     return -EINVAL;
70
71   if (uv__is_active(handle))
72     uv_timer_stop(handle);
73
74   clamped_timeout = handle->loop->time + timeout;
75   if (clamped_timeout < timeout)
76     clamped_timeout = (uint64_t) -1;
77
78   handle->timer_cb = cb;
79   handle->timeout = clamped_timeout;
80   handle->repeat = repeat;
81   /* start_id is the second index to be compared in uv__timer_cmp() */
82   handle->start_id = handle->loop->timer_counter++;
83
84   heap_insert((struct heap*) &handle->loop->timer_heap,
85               (struct heap_node*) &handle->heap_node,
86               timer_less_than);
87   uv__handle_start(handle);
88
89   return 0;
90 }
```

- L68-L69, 做参数的检查，错误则返回 -EINVAL。
- L71-L72，如有是一个活跃的 timer, 则立即停止它。
- L74-L82, 参数赋值，上面提到的 `start_id` 就是由 `timer_counter` 自增得到。
- L84-L86, 插入 timer 节点到最小堆，此处算法复杂度为 O(lgn)。
- L87， 标记句柄非活跃，并加入统计。

```
93 int uv_timer_stop(uv_timer_t* handle) {
94   if (!uv__is_active(handle))
95     return 0;
96
97   heap_remove((struct heap*) &handle->loop->timer_heap,
98               (struct heap_node*) &handle->heap_node,
99               timer_less_than);
100  uv__handle_stop(handle);
101
102  return 0;
103 }
```

L94，检查 handle, 如果是非获取的，则说明没有启动过，则返回成功。 L97-L99, 从最小堆中删除 timer的节点。 L100, 重置句柄，并减少计数。

了解了如何开启和关闭一个定时器，我们看如何调度定时器。

```
int uv_run(uv_loop_t* loop, uv_run_mode mode) {
  ...
  while (r != 0 && loop->stop_flag == 0) {
    uv__update_time(loop);
    uv__run_timers(loop);
    ran_pending = uv__run_pending(loop);
    ...
 }
```

在 node.js 的 event loop 中，更新时间后则立即调用 `uv__run_timers` ，可见 timer 作为一个外部系统依赖的模块，优先级是最高的。

```c
150 void uv__run_timers(uv_loop_t* loop) {
151   struct heap_node* heap_node;
152   uv_timer_t* handle;
153
154   for (;;) {
155     heap_node = heap_min((struct heap*) &loop->timer_heap);
156     if (heap_node == NULL)
157       break;
158
159     handle = container_of(heap_node, uv_timer_t, heap_node);
160     if (handle->timeout > loop->time)
161       break;
162
163     uv_timer_stop(handle);
164     uv_timer_again(handle);
165     handle->timer_cb(handle);
166   }
167 }
```

L155-L157, 取出最小的timer节点,如果为空，则跳出循环。 L159-L161, 通过 heap_node 的偏移拿到对象的首地址，如果最小的 timeout时间大于当前的时间，则说明过期时间还没到，则退出循环。 L163-L165, 删除 timer, 如果是需要重复执行的定时器，则通过调用 `uv_timer_again` 再次加入, L165执行 timer的 callback 任务后循环。

改进的分级时间轮实现

https://github.com/libuv/libuv/pull/823

桥接层

阅读此节需要node.js addon 的知识，这边默认你已经了解。

```
    env->SetProtoMethod(constructor, "start", Start);
    env->SetProtoMethod(constructor, "stop", Stop);

    target->Set(FIXED_ONE_BYTE_STRING(env->isolate(), "Timer"),
                constructor->GetFunction());
```

Timer 的 addon 导出 `start` , `stop` 的方法，供js 层调用。

```
71  static void Start(const FunctionCallbackInfo<Value>& args) {
72    TimerWrap* wrap = Unwrap<TimerWrap>(args.Holder());
73
74    CHECK(HandleWrap::IsAlive(wrap));
75
76    int64_t timeout = args[0]->IntegerValue();
77    int64_t repeat = args[1]->IntegerValue();
78    int err = uv_timer_start(&wrap->handle_, OnTimeout, timeout, repeat);
79    args.GetReturnValue().Set(err);
80  }
81
82  static void Stop(const FunctionCallbackInfo<Value>& args) {
83    TimerWrap* wrap = Unwrap<TimerWrap>(args.Holder());
84
85    CHECK(HandleWrap::IsAlive(wrap));
86
87    int err = uv_timer_stop(&wrap->handle_);
88    args.GetReturnValue().Set(err);
89  }
```

`Start` 需要提供两个参数，1.超时时间 timeout; 2. 重复执行的周期。 L78 调用 `uv_timer_start` ,其中 `OnTimeout` 是该定时器的回调函数。 我们看下该函数实现：

```
91   static void OnTimeout(uv_timer_t* handle) {
92     TimerWrap* wrap = static_cast<TimerWrap*>(handle->data);
93     Environment* env = wrap->env();
94     HandleScope handle_scope(env->isolate());
95     Context::Scope context_scope(env->context());
96     wrap->MakeCallback(kOnTimeout, 0, nullptr);
97   }
```

你可能好奇，怎么就由 handle->data 取到对象指针了呢？

```
HandleWrap::HandleWrap(Environment* env,
                       Local<Object> object,
                       uv_handle_t* handle,
                       AsyncWrap::ProviderType provider,
                       AsyncWrap* parent)
    : AsyncWrap(env, object, provider, parent),
      flags_(0),
      handle__(handle) {
  handle__->data = this;
  ...
}
```

由于 `TimerWrap` 继承自 `HandleWrap`，对象构造时就把 `handle` 的私有变量 `data` 指向了 this 指针，也就是 `HandleWrap`。回调函数通过强转获取了 `TimerWrap` 对象。

令人感兴趣的是 L96,这边是由 C++ 调用 jsland. 查看该处的修改历史，笔者发现：

> timers: dispatch ontimeout callback by array index
>
> Achieve a minor speed-up by looking up the timeout callback on the timer object by using an array index rather than a named property.
>
> Gives a performance boost of about 1% on the misc/timers benchmarks.

之前的实现是属性查找，而通过极致的优化，属性查找被替换成数组索引，benchmark性能提升了1%。 而整个系统性能的提升正是来源于这点滴的积累。

timers.js

有了桥接层，js便有了开启、关闭一个定时器的能力。

为了不影响到nodejs中的event loop，timer模块专门提供了一些内部的api: `timers._unrefActive` 给像socket这样的对象使用。

在最初的设计中，每次执行_unrefActive添加任务时都会维持着unrefList的顺序，保证超时时间最小的处于前面。这样在定时器超时后便可以以最快的速度处理超时任务并设置下一个定时器，但是在添加任务时最坏的情况下需要遍历unrefList链表中的所有节点。

```
exports._unrefActive = function(item) {
  var msecs = item._idleTimeout;
  if (!msecs || msecs < 0) return;
  assert(msecs >= 0);

  L.remove(item);

  if (!unrefList) {
    debug('unrefList initialized');
    unrefList = {};
    L.init(unrefList);

    debug('unrefTimer initialized');
    unrefTimer = new Timer();
    unrefTimer.unref();
    unrefTimer.when = -1;
    unrefTimer[kOnTimeout] = unrefTimeout;
  }

  var now = Timer.now();
  item._idleStart = now;

  if (L.isEmpty(unrefList)) {
    debug('unrefList empty');
    L.append(unrefList, item);

    unrefTimer.start(msecs, 0);
    unrefTimer.when = now + msecs;
    debug('unrefTimer scheduled');
    return;
```

```
547   }
548
549   var when = now + msecs;
550
551   debug('unrefList find where we can insert');
552
553   var cur, them;
554
555   for (cur = unrefList._idlePrev; cur != unrefList; cur = cur._idlePrev) {
556     them = cur._idleStart + cur._idleTimeout;
557
558     if (when < them) {
559       debug('unrefList inserting into middle of list');
560
561       L.append(cur, item);
562
563       if (unrefTimer.when > when) {
564         debug('unrefTimer is scheduled to fire too late, reschedule');
565         unrefTimer.start(msecs, 0);
566         unrefTimer.when = when;
567       }
568
569       return;
570     }
571   }
572
573   debug('unrefList append to end');
574   L.append(unrefList, item);
575 };
```

L524-L534, 是有且只创建一个 `unrefTimer` ,来处理超时的内部使用定时器，处理完一个则顺序处理下一个。

L553-L571, 当需要插入一个定时器时，则需要保证 `unrefList` 有序，需要遍历链表找到插入的位置，最差的情况下是 O(N)。

很显然，在HTTP中建立连接是最频繁的操作，那么向 `unrefList` 链表中添加节点也就非常频繁了，而且最开始设置的定时器其实最后真正会超时的非常少，因为中间涉及到io的正常操作时便会取消定时器。所以问题就变成最耗性能的操作非常频繁，而几乎不花时间的操作却很少被执行到。

针对这种情况，如何解决呢？

显然这里也遵从80/20原则。思路上我们应该使80%的情况变得更高效。

使用不排序的链表

主要思路就是将对unrefList链表的遍历操作，移到unrefTimeout定时器超时处理中。这样每次查找出已经超时的任务就需要花比较多的时间了O(n)，但是插入操作却变得非常简单O(1)，而插入节点正是最频繁的操作。

```
572 exports._unrefActive = function(item) {
573   ....省略
574   var now = Timer.now();
575   item._idleStart = now;
576
577   var when = now + msecs;
578
579   // If the actual timer is set to fire too late, or not set to fire at all,
580   // we need to make it fire earlier
581   if (unrefTimer.when === -1 || unrefTimer.when > when) {
582     unrefTimer.start(msecs, 0);
583     unrefTimer.when = when;
584     debug('unrefTimer scheduled');
585   }
586
587   debug('unrefList append to end');
588   L.append(unrefList, item);
589 };
```

可以看到 L588，之前遍历查找在新的实现中 e5bb668，简单的变成抽象List的 `append` 操作。

> https://github.com/joyent/node/issues/8160

使用二叉堆

二叉堆达到了插入和查找的平衡，和目前 libuv 的实现一致。 有兴趣的可以查看：

- https://github.com/misterdjules/node/commits/fix-issue-8160-with-heap, 基于 v0.12.

社区改进实现

- 有序链表的实现的版本只采用了一个 `unrefTimer` 来执行任务，在内存上是节省了，但却很难达到性能的平衡。
- 二叉堆实现在正常的连接场景下却输于不排序链表。

社区通过演变，实现采用的是哈希+链表的结合，以空间换时间。其实是一种时间轮算法的演化。

```
╔════ > Object Map
║
╠══
║ refedLists: { '40': { }, '320': { etc } } (keys of millisecond duration)
╚══          ┌──────────┘
             │
╔══          │
║ TimersList { _idleNext: { }, _idlePrev: (self), _timer: (TimerWrap) }
║         ┌────────────────┘
║    ╔══  │                              ^
║    ║    { _idleNext: { },  _idlePrev: { }, _onTimeout: (callback) }
║    ║      ┌───────────┘
║    ║      │                                  ^
║    ║      { _idleNext: { etc },  _idlePrev: { }, _onTimeout: (callback) }
╠══  ╠══
║    ║
║    ╚════ >  Actual JavaScript timeouts
║
╚════ > Linked List
```

我们先看下数据结构的组织：

- `refedLists` 的键是超时时间，值是一个具有相同超时时间的链表。
- `unrefedLists` 也是同理。

```
// Internal APIs that need timeouts should use `_unrefActive()` instead of
// `active()` so that they do not unnecessarily keep the process open.
exports._unrefActive = function(item) {
  insert(item, true);
};
// The underlying logic for scheduling or re-scheduling a timer.
//
// Appends a timer onto the end of an existing timers list, or creates a new
// TimerWrap backed list if one does not already exist for the specified timeout
// duration.
function insert(item, unrefed) {
  const msecs = item._idleTimeout;
  if (msecs < 0 || msecs === undefined) return;

  item._idleStart = TimerWrap.now();

  const lists = unrefed === true ? unrefedLists : refedLists;

  // Use an existing list if there is one, otherwise we need to make a new one.
  var list = lists[msecs];
  if (!list) {
    debug('no %d list was found in insert, creating a new one', msecs);
    // Make a new linked list of timers, and create a TimerWrap to schedule
    // processing for the list.
    list = new TimersList(msecs, unrefed);
    L.init(list);
```

```
135     list._timer._list = list;
136
137     if (unrefed === true) list._timer.unref();
138     list._timer.start(msecs, 0);
139
140     lists[msecs] = list;
141     list._timer[kOnTimeout] = listOnTimeout;
142   }
143
144   L.append(list, item);
145   assert(!L.isEmpty(list)); // list is not empty
146 }
```

我们比较下上述实现：

- L128，根据键值（超时时间）拿到 list ，如有不为undefined,则简单的 `append` 到最后面就好了,复杂度O(1)。
- L130-L141, 如果为 undefined, 则创建一个 `TimersList` ,包含一个C的定时器，来处理链表中的任务。
- `listOnTimeout` 也变得很简单，取出链表的任务，复杂度取决于链表的长度O(m), m < N。

模块使用一个链表来保存所有超时时间相同的对象，每个对象中都会存储开始时间_idleStart以及超时时间_idleTimeout。链表中第一个加入的对象一定会比后面加入的对象先超时，当第一个对象超时完成处理后，重新计算下一个对象是否已经到时或者还有多久到时，之前创建的Timer对象便会再次启动并设置新的超时时间，直到当链表上所有的对象都已经完成超时处理，此时便会关闭这个Timer对象。

通过这种巧妙的设计，使得一个Timer对象得到了最大的复用，从而极大的提升了timer模块的性能。

Timer在node中的应用

- 动态更新 HTTP Date字段的缓存

```
31 var dateCache;
32 function utcDate() {
33   if (!dateCache) {
34     var d = new Date();
35     dateCache = d.toUTCString();
36     timers.enroll(utcDate, 1000 - d.getMilliseconds());
37     timers._unrefActive(utcDate);
38   }
39   return dateCache;
40 }
41 utcDate._onTimeout = function() {
42   dateCache = undefined;
43 };

228   // Date header
229   if (this.sendDate === true && state.sentDateHeader === false) {
230     state.messageHeader += 'Date: ' + utcDate() + CRLF;
231   }
```

L230，每次构造 Date字段值都会去获取系统时间，但精度要求不高，只需要秒级就够了，所以在1S 的连接请求可以复用 dateCache 的值，超时后重置为 `undefined` .

L34-L35,下次获取会重启生成。

L36-L37,重新设置超时时间以便更新。

- HTTP 连接超时控制

```
303   if (self.timeout)
304     socket.setTimeout(self.timeout);
305   socket.on('timeout', function() {
306     var req = socket.parser && socket.parser.incoming;
307     var reqTimeout = req && !req.complete && req.emit('timeout', socket);
308     var res = socket._httpMessage;
309     var resTimeout = res && res.emit('timeout', socket);
310     var serverTimeout = self.emit('timeout', socket);
311
312     if (!reqTimeout && !resTimeout && !serverTimeout)
313       socket.destroy();
314   });
```

默认的 timeout 为 `this.timeout = 2 * 60 * 1000;` 也就是120s。 L313，超时则销毁 socket。

小结

Node.js 的 timer 模块闪烁着很多程序设计的精髓。

- 数据结构抽象
  - linkedlist.js 抽象出链表的基础操作。
- 以空间换时间
  - 相同超时时间的定时器分组，而不是使用一个 `unrefTimer` ，复杂度降到 O(1)。
- 对象复用
  - 相同超时时间的定时器共享一个底层的 C的 timer。
- 80/20法则
  - 优化主要路径的性能。

参考文档

[1].https://github.com/nodejs/node/wiki/Optimizing-_unrefActive

[2].http://alinode.aliyun.com/blog/9

Yield 魔法

ES6中的Generator的引入，极大程度上改变了JavaScript程序员对迭代器的看法，并为解决 `callback hell` 提供了新方法。

Generators

迭代器模式是很常用的设计模式，但是实现起来，很多东西是程序化的；当迭代规则比较复杂时，维护迭代器内的状态，是比较麻烦的。 于是有了generator，何为generator？

> Generators: a better way to build Iterators.

借助 yield 关键字,可以更优雅的实现fibonacci数列。

```
function* fibonacci() {
  let a = 0, b = 1;

  while(true) {
    yield a;
    [a, b] = [b, a + b];
  }
}
```

yield与异步

yield可以暂停运行流程，那么便为改变执行流程提供了可能。这和Python的coroutine类似。

Geneartor之所以可用来控制代码流程，就是通过yield来将两个或者多个Geneartor的执行路径互相切换。这种切换是语句级别的，而不是函数调用级别的。其本质是CPS变换。

yield之后，实际上本次调用就结束了，控制权实际上已经转到了外部调用了generator的next方法的函数，调用的过程中伴随着状态的改变。那么如果外部函数不继续调用next方法，那么yield所在函数就相当于停在yield那里了。所以把异步的

东西做完，要函数继续执行，只要在合适的地方再次调用generator 的next就行，就好像函数在暂停后，继续执行。

V8 实现

parse phase

Generator function 和 `yield` 关键字处理是在 `parser.cc`，我们看到 AST 解析函数： `Parser::ParseEagerFunctionBody()`

```
ZoneList<Statement*>* Parser::ParseEagerFunctionBody(
    const AstRawString* function_name, int pos, Variable* fvar,
    Token::Value fvar_init_op, FunctionKind kind, bool* ok) {
    .....
  // For generators, allocate and yield an iterator on function entry.
  if (IsGeneratorFunction(kind)) {
    ZoneList<Expression*>* arguments =
        new(zone()) ZoneList<Expression*>(0, zone());
    CallRuntime* allocation = factory()->NewCallRuntime(
        ast_value_factory()->empty_string(),
        Runtime::FunctionForId(Runtime::kCreateJSGeneratorObject), arguments,
        pos);
    VariableProxy* init_proxy = factory()->NewVariableProxy(
        function_state_->generator_object_variable());
    Assignment* assignment = factory()->NewAssignment(
        Token::INIT_VAR, init_proxy, allocation, RelocInfo::kNoPosition);
    VariableProxy* get_proxy = factory()->NewVariableProxy(
        function_state_->generator_object_variable());
    Yield* yield = factory()->NewYield(
        get_proxy, assignment, Yield::kInitial, RelocInfo::kNoPosition);
    body->Add(factory()->NewExpressionStatement(
        yield, RelocInfo::kNoPosition), zone());
```

```
3972   }
3973
3974   ParseStatementList(body, Token::RBRACE, false, NULL, CHECK_OK);
3975
3976   if (IsGeneratorFunction(kind)) {
3977     VariableProxy* get_proxy = factory()->NewVariableProxy(
3978         function_state_->generator_object_variable());
3979     Expression* undefined =
3980         factory()->NewUndefinedLiteral(RelocInfo::kNoPosition);
3981     Yield* yield = factory()->NewYield(get_proxy, undefined, Yield::kFinal,
3982                                        RelocInfo::kNoPosition);
3983     body->Add(factory()->NewExpressionStatement(
3984         yield, RelocInfo::kNoPosition), zone());
3985   }
3986    ...
```

L3955 判断是否是Generator function。 `ParseStatementList` 解析 function 函数体。 注意，Generator function 也是一种 function, 在 V8中，同样用 `JSFunction` 表示。

在两个 if 函数体中，创建了 `Yield::kInitial` 和 `Yield::kFinal` 两个Yield AST 节点。

Yield 状态分为:

```
enum Kind {
    kInitial,  // The initial yield that returns the unboxed generator object.
    kSuspend,  // A normal yield: { value: EXPRESSION, done: false }
    kDelegating,  // A yield*.
    kFinal        // A return: { value: EXPRESSION, done: true }
  };
```

codegen phase

机器码生成(x64平台)主要集中在 `runtime-generator.cc` , `full-codegen-x64.cc` 。

`runtime-generator.cc` 提供了 `Create` , `Suspend` , `Resume` , `Close` 等 stub 代码段，

给 full-codegen 内联使用，生成汇编代码。

我们先来看到 `RUNTIME_FUNCTION(Runtime_CreateJSGeneratorObject)` ,

```
RUNTIME_FUNCTION(Runtime_CreateJSGeneratorObject) {
  HandleScope scope(isolate);
  DCHECK(args.length() == 0);

  JavaScriptFrameIterator it(isolate);
  JavaScriptFrame* frame = it.frame();
  Handle<JSFunction> function(frame->function());
  RUNTIME_ASSERT(function->shared()->is_generator());

  Handle<JSGeneratorObject> generator;
  if (frame->IsConstructor()) {
    generator = handle(JSGeneratorObject::cast(frame->receiver()));
  } else {
    generator = isolate->factory()->NewJSGeneratorObject(function);
  }
  generator->set_function(*function);
  generator->set_context(Context::cast(frame->context()));
  generator->set_receiver(frame->receiver());
  generator->set_continuation(0);
  generator->set_operand_stack(isolate->heap()->empty_fixed_array());
  generator->set_stack_handler_index(-1);

  return *generator;
}
```

函数根据当前的 Frame, 创建一个 `JSGeneratorObject` 对象来储存 `JSFunction` , `Context` ，pc 指针， 设置操作数栈为空。

yield 后，实际上就是保存当前的执行环境，L74保存当前的操作数栈，并保存到JSGeneratorObject对象中。

```
40 RUNTIME_FUNCTION(Runtime_SuspendJSGeneratorObject) {
41   HandleScope handle_scope(isolate);
42   DCHECK(args.length() == 1);
43   CONVERT_ARG_HANDLE_CHECKED(JSGeneratorObject, generator_object, 0);
44
45   JavaScriptFrameIterator stack_iterator(isolate);
46   JavaScriptFrame* frame = stack_iterator.frame();
47   RUNTIME_ASSERT(frame->function()->shared()->is_generator());
48   DCHECK_EQ(frame->function(), generator_object->function());
49
50   // The caller should have saved the context and continuation already.
51   DCHECK_EQ(generator_object->context(), Context::cast(frame->context()));
52   DCHECK_LT(0, generator_object->continuation());
53
54   // We expect there to be at least two values on the operand stack: the return
55   // value of the yield expression, and the argument to this runtime call.
56   // Neither of those should be saved.
57   int operands_count = frame->ComputeOperandsCount();
58   DCHECK_GE(operands_count, 2);
59   operands_count -= 2;
60
61   if (operands_count == 0) {
62     // Although it's semantically harmless to call this function with an
63     // operands_count of zero, it is also unnecessary.
64     DCHECK_EQ(generator_object->operand_stack(),
65               isolate->heap()->empty_fixed_array());
```

```
66      DCHECK_EQ(generator_object->stack_handler_index(), -1);
67      // If there are no operands on the stack, there shouldn't be a handler
68      // active either.
69      DCHECK(!frame->HasHandler());
70    } else {
71      int stack_handler_index = -1;
72      Handle<FixedArray> operand_stack =
73          isolate->factory()->NewFixedArray(operands_count);
74      frame->SaveOperandStack(*operand_stack, &stack_handler_index);
75      generator_object->set_operand_stack(*operand_stack);
76      generator_object->set_stack_handler_index(stack_handler_index);
77    }
78
79    return isolate->heap()->undefined_value();
80  }
```

Resume 对应于外部的 `next` ，要恢复执行，首先我们得知道需要执行的 pc 指针偏移，机器代码存储在 `JSFunction` 的 `Code` 对象中，L105 拿到 pc 首地址，L106从 `JSGeneratorObject` 对象 取出偏移 offset 。

L108 设置当前 Frame 的 pc 偏移。L118 恢复操作数栈，L126-L130根据恢复的 mode，返回 value。

```
90 RUNTIME_FUNCTION(Runtime_ResumeJSGeneratorObject) {
91    SealHandleScope shs(isolate);
92    DCHECK(args.length() == 3);
93    CONVERT_ARG_CHECKED(JSGeneratorObject, generator_object, 0);
94    CONVERT_ARG_CHECKED(Object, value, 1);
95    CONVERT_SMI_ARG_CHECKED(resume_mode_int, 2);
96    JavaScriptFrameIterator stack_iterator(isolate);
97    JavaScriptFrame* frame = stack_iterator.frame();
98
99    DCHECK_EQ(frame->function(), generator_object->function());
100   DCHECK(frame->function()->is_compiled());
```

```

  STATIC_ASSERT(JSGeneratorObject::kGeneratorExecuting < 0);
  STATIC_ASSERT(JSGeneratorObject::kGeneratorClosed == 0);

  Address pc = generator_object->function()->code()->instruction_start();
  int offset = generator_object->continuation();
  DCHECK(offset > 0);
  frame->set_pc(pc + offset);
  ...
  generator_object->set_continuation(JSGeneratorObject::kGeneratorExecuting);

  FixedArray* operand_stack = generator_object->operand_stack();
  int operands_count = operand_stack->length();
  if (operands_count != 0) {
    frame->RestoreOperandStack(operand_stack,
                               generator_object->stack_handler_index());
    generator_object->set_operand_stack(isolate->heap()->empty_fixed_array());
    generator_object->set_stack_handler_index(-1);
  }

  JSGeneratorObject::ResumeMode resume_mode =
      static_cast<JSGeneratorObject::ResumeMode>(resume_mode_int);
  switch (resume_mode) {
    case JSGeneratorObject::NEXT:
      return value;
    case JSGeneratorObject::THROW:
      return isolate->Throw(value);
  }
  ...
}
```

这边我们关注下 args 参数， args[0]是 `JSGeneratorObject` 对象 `generator_object` ， args[1]是Object 对象 `value` , 也就是 `next` 的返回值，args[2]是表示 resume 模式的值。

对应的我们看到 `FullCodeGenerator::EmitGeneratorResume()` 中的这几行代码：

```
2296    __ Push(rbx);
2297    __ Push(result_register());
2298    __ Push(Smi::FromInt(resume_mode));
2299    __ CallRuntime(Runtime::kResumeJSGeneratorObject, 3);
```

L2297从 result 寄存器中取出 value, L2299调用 `RUNTIME_FUNCTION(Runtime_ResumeJSGeneratorObject)` 。

这样，从 yield value 到 g.next() 取出 value, 相信大家有了一个大概的认知了。

延伸

我们看到node.js依托 v8层面实现了协程，有兴趣的同学可以关心下 fibjs, 它是用 C 库实现了协程，遇到异步调用就 "yield" 放弃 CPU， 交由协程调度，也解决了 callcack hell 的问题。 本质思想上两种方案没本质区别：

- Generator是利用yield特殊关键字来暂停执行，而fibers是利用Fiber.yield()暂停
- Generator是利用函数返回的Generator句柄来控制函数的继续执行，而fibers是在异步回调中利用Fiber.current.run()继续执行。

参考

- http://en.wikipedia.org/wiki/Continuation-passing_style
- https://zh.wikipedia.org/zh-cn/协程
- fibjs https://github.com/xicilion/fibjs

Buffer

在Node.js中，Buffer类是随Node内核一起发布的核心库。Buffer库为Node.js带来了一种存储原始数据的方法，可以让Nodejs处理二进制数据，每当需要在Nodejs中处理I/O操作中移动的数据时，就有可能使用Buffer库。原始数据存储在 Buffer 类的实例中。一个 Buffer 类似于一个整数数组，但它对应于 V8 堆内存之外的一块原始内存。

Buffer 和 Javascript 字符串对象之间的转换需要显式地调用编码方法来完成。以下是几种不同的字符串编码：

- ‘ascii’ – 仅用于 7 位 ASCII 字符。这种编码方法非常快，并且会丢弃高位数据。
- ‘utf8’ – 多字节编码的 Unicode 字符。许多网页和其他文件格式使用 UTF-8。
- ‘ucs2’ – 两个字节，以小尾字节序(little-endian)编码的 Unicode 字符。它只能对 BMP（基本多文种平面，U+0000 – U+FFFF） 范围内的字符编码。
- ‘base64’ – Base64 字符串编码。
- ‘binary’ – 一种将原始二进制数据转换成字符串的编码方式，仅使用每个字符的前 8 位。这种编码方法已经过时，应当尽可能地使用 Buffer 对象。
- 'hex' - 每个字节都采用 2 进制编码。

在Buffer中创建一个数组，需要注意以下规则：

- Buffer 是内存拷贝，而不是内存共享。

- Buffer 占用内存被解释为一个数组，而不是字节数组。比如，new Uint32Array(new Buffer([1,2,3,4])) 创建了4个 Uint32Array，它的成员为 [1,2,3,4] ,而不是[0x1020304] 或 [0x4030201]。

slab 分配

在 lib/buffer.js 模块中，有个模块私有变量 pool， 它指向当前的一个8K 的slab ：

```
Buffer.poolSize = 8 * 1024;
var pool;

function allocPool() {
  pool = new SlowBuffer(Buffer.poolSize);
  pool.used = 0;
}
```

SlowBuffer 为 src/node_buffer.cc 导出，当用户调用new Buffer时 ，如果你要申请的空间大于8K，node 会直接调用SlowBuffer ，如果小于8K ，新的Buffer 会建立在当前slab 之上：

- 新创建的Buffer的 parent成员变量会指向这个slab ，
- offset 变量指向在这个slab 中的偏移：

  ```
  if (!pool || pool.length - pool.used < this.length) allocPoo
  l();
  this.parent = pool;
  this.offset = pool.used;
  pool.used += this.length;
  ```

PS: 在 lib/_tls_legacy.js 中， `SlabBuffer` 创建了一个 10MB 的 slab。

```
function alignPool() {
  // Ensure aligned slices
  if (poolOffset & 0x7) {
   poolOffset |= 0x7;
   poolOffset++;
  }
}
```

这里做了8字节的内存对齐处理。

- 如果不按照平台要求对数据存放进行对齐，会带来存取效率上的损失。比如32位的Intel处理器通过总线访问(包括读和写)内存数据。每个总线周期从偶地址开始访问32位内存数据，内存数据以字节为单位存放。如果一个32位的数据没有存放在4字节整除的内存地址处，那么处理器就需要2个总线周期对其进行访问，显然访问效率下降很多。

- Node.js 是一个跨平台的语言，第三方的C++ addon 也是非常多，避免破坏了第三方模块的使用，比如 directIO 就必须要内存对齐。
- 兼容 node.js v0.10

> 详细：https://github.com/nodejs/node/pull/2487

浅拷贝

Buffer更像是可以做指针操作的C语言数组。例如，可以用[index]方式直接修改某个位置的字节。 需要注意的是：Buffer#slice 方法， 不是返回一个新的Buffer， 而是返回对原 Buffer 某个区间数值的引用。

```
const buf1 = Buffer.allocUnsafe(26);

for (var i = 0 ; i < 26 ; i++) {
  buf1[i] = i + 97; // 97 is ASCII a
}

const buf2 = buf1.slice(0, 3);
buf2.toString('ascii', 0, buf2.length);
  // Returns: 'abc'
buf1[0] = 33;
buf2.toString('ascii', 0, buf2.length);
  // Returns : '!bc'
```

上面是官方 API 提供的例子， `buf2` 是对 `buf1` 前3个字节的引用，对 `buf2` 的修改就相当于作用在 `buf1` 上。

深拷贝

如果想要拷贝一份Buffer，得首先创建一个新的Buffer，并通过.copy方法把原Buffer中的数据复制过去。

```
const buf1 = Buffer.allocUnsafe(26);
const buf2 = Buffer.allocUnsafe(26).fill('!');

for (let i = 0 ; i < 26 ; i++) {
  buf1[i] = i + 97; // 97 is ASCII a
}

buf1.copy(buf2, 8, 16, 20);
console.log(buf2.toString('ascii', 0, 25));
  // Prints: !!!!!!!!qrst!!!!!!!!!!!!!
```

通过深拷贝的方式， `buf2` 截取了 `buf1` 的部分内容，之后对 `buf2` 的修改并不会作用于 `buf1` , 两者内容独立不共享。

需要注意的事：深拷贝是一种消耗 CPU 和内存的操作，请知道自己在做什么。

内存碎片

动态分配将不可避免会产生内存碎片的问题，那么什么是内存碎片？ 内存碎片即“碎片的内存”描述一个系统中所有不可用的空闲内存，这些碎片之所以不能被使用，是因为负责动态分配内存的分配算法使得这些空闲的内存无法使用。

上述的 slab 分配，存在明显的内存碎片，即 8KB 的内存并没有完全被使用，存在一定的浪费。通用的slab实现，会浪费约1/2的空间。

当然存在更高效，更省内存的内存管理分配，比如 tcmalloc, 但也必须承受一定的管理代价。node.js 在这方面并没有一味的执着于此，而是达到一种性能与空间使用的平衡。

zero fill

Node.js 在v5.10.0 加入了命令行选项 `--zero-fill-buffers` , 强制在申请 `Buffer` 时用0填充分配的内存。

为什么要引入这个特性呢？

- 防止你代码中本该初始化的地方没有初始化；
- 防止其他代码访问到你之前写入 Buffer 的数据， 这边存在安全隐患, 如下

```
✗ node -p "new Buffer(1024).toString('ascii')"
`7(@ P
xn?_k7x0x0' @#k
            :ArrayBuffer kh
                           &7;?m@bFn?_ @`` n?0'h2R'Lq083~C[e
;@string k (R!~!H3kl
```

代码实现上则是通过 `--zero-fill-buffers` 区分申请内存是用 `malloc()` 或者 `calloc()` 。

当然性能上 `calloc()` 还是差很多，所以社区开放了一个选项，而不是默认开启。

- here are benchmark results for allocating a 1mb Buffer:

xxx	v5.4.0	v4.2.3	v0.12.9	v0.10.41
new Buffer	41,515 ops/sec ±3.00%	43,670 ops/sec ±1.86%	53,969 ops/sec ±1.41%	147,350 ops/sec ±1.82%
new Buffer (zero-filled)	5,041 ops/sec ±2.00%	4,293 ops/sec ±1.79%	7,953 ops/sec ±0.55%	8,167 ops/sec ±2.38%

> 具体了解：https://github.com/nodejs/node/issues/4660

总结

Buffer是一个典型的Javascript和C++结合的模块，性能相关部分用C++实现，非性能相关部分用javascript实现。

Node在进程启动时Buffer就已经加装进入内存，并将其放入全局对象，因此无需require。

Buffer内存分配，Buffer对象的内存分配不是在V8的堆内存中，在Node的C++层面实现内存的申请。

参考

- https://nodesource.com/blog/nsolid-deepdive-into-security-policies-zero-fill-

buffer-allocations/

Event

> Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient.

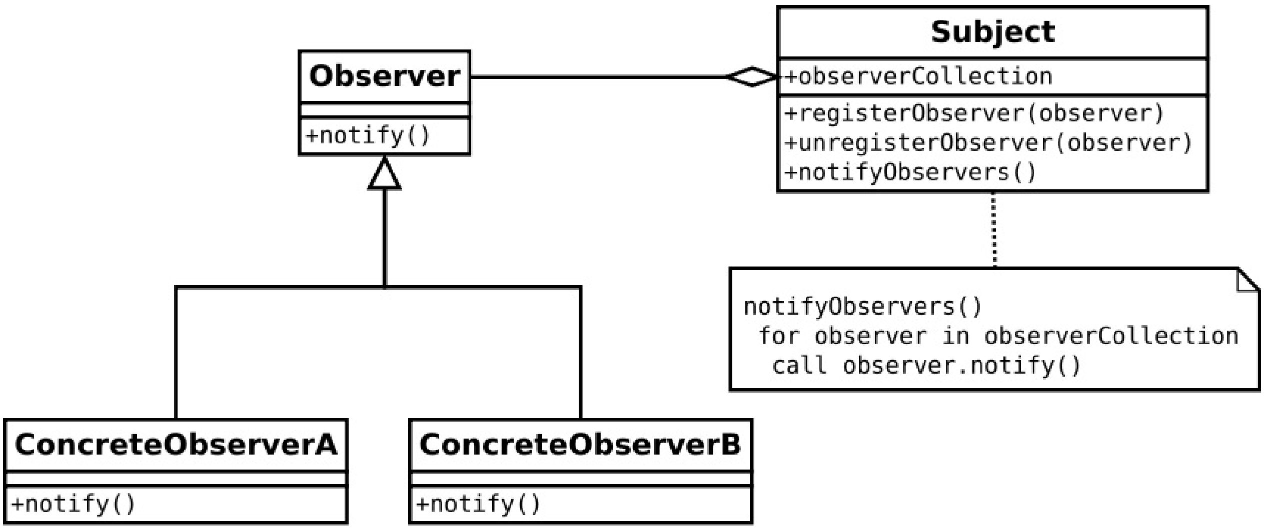
这是Node.Js官网对自身的介绍,明确强调了Node.Js使用了一个事件驱动、非阻塞式 I/O 的模型,使其轻量又高效。

而且在Node中大量核心模块都使用了Event的机制,因此可以说是整个Node里最重要的模块之一.

涉及源码

- lib/events.js

观察者模式



上图是 UML 的类图，

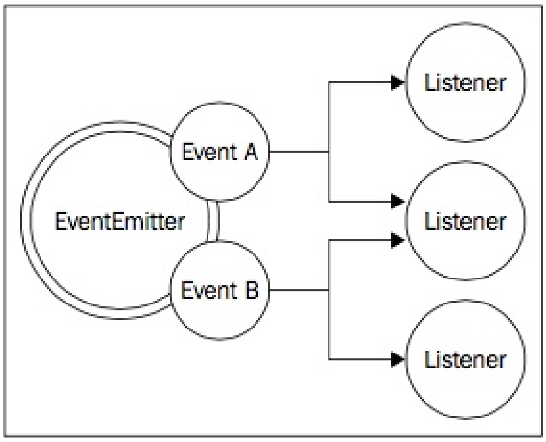
观察者模式是这样一种设计模式。一个被称作被观察者的对象，维护一组被称为观察者的对象，这些对象依赖于被观察者，被观察者自动将自身的状态的任何变化通知给它们。

当一个被观察者需要将一些变化通知给观察者的时候，它将采用广播的方式，这条广播可能包含特定于这条通知的一些数据。

使用观察者模式更深层次的动机是，当我们需要维护相关对象的一致性的时候，我们可以避免对象之间的紧密耦合。例如，一个对象可以通知另外一个对象，而不需要知道这个对象的信息。

Event.js 实现

EventEmitter 允许我们注册一个或多个函数作为 listeners。 在特定的事件触发时被调用。如下图：



listeners 存储

一般观察者的设计模式的实现逻辑是类似的，都是有一个类似map的结构，存储监听事件和回调函数的对应关系。

```
// This constructor is used to store event handlers. Instantiating this is
// faster than explicitly calling `Object.create(null)` to get a "clean" empty
// object (tested with v8 v4.9).
function EventHandlers() {}
EventHandlers.prototype = Object.create(null);
EventEmitter.init = function() {
  ...
  if (!this._events || this._events === Object.getPrototypeOf(this)._events) {
    this._events = new EventHandlers();
    this._eventsCount = 0;
  }

  this._maxListeners = this._maxListeners || undefined;
};
```

在 EventEmitter 类中，以 键 / 值 对的方式来存储事件名和对应的监听器。 你可以会好奇，为什么创建一个最简单的 键 / 值 对搞的这么复杂，简单的一个 `this._events = {};` 不就好咯。

是的，社区的最初实现是这样的，但随着 V8的升级，对 ES6支持的越来越完备，它的实现办法是使用一个空的构造函数，并且把这个构造的原型事先置空。

通过jsperf 比较两者的性能, 我们发现这种实现竟是简单实现性能的2倍！

增加事件监听

addListener: 增加事件监听, on： addListener的别名，实际上是一样的。

```
function _addListener(target, type, listener, prepend) {
  var m;
  var events;
  var existing;

  if (typeof listener !== 'function')
    throw new TypeError('"listener" argument must be a function');
```

```
217
218   events = target._events;
219   if (!events) {
220     events = target._events = new EventHandlers();
221     target._eventsCount = 0;
222   } else {
223     ...
234   }
235
236   if (!existing) {
237     // Optimize the case of one listener. Don't need the extra array object.
238     existing = events[type] = listener;
239     ++target._eventsCount;
240   } else {
241     if (typeof existing === 'function') {
242       // Adding the second element, need to change to array.
243       existing = events[type] = prepend ? [listener, existing] :
244                                           [existing, listener];
245     } else {
246       // If we've already got an array, just append.
247       if (prepend) {
248         existing.unshift(listener);
249       } else {
250         existing.push(listener);
251       }
252     }
253
254     // Check for listener leak
255     ...
264   }
265
266   return target;
267 }
```

实际使用复杂场景时，会出现对回调顺序的需求。L250,默认添加监听是在事件监听数组的末尾。L247-L248， `prepend` 标记是否在事件数组的前部添加。

> 深入了解 https://github.com/nodejs/node/pull/6032

删除事件监听

在 EventEmitter#removeListener 这个 API 的实现里，需要从存储的监听器数组中除去一个元素，我们首先想到的就是使用 Array#splice 这个 API ，即 arr.splice(i, 1) 。不过这个 API 所提供的功能过于多了，它支持去除自定义数量的元素，还支持向数组中添加自定义的元素。所以，源码中选择自己实现一个最小可用的：

```
function spliceOne(list, index) {
  for (var i = index, k = i + 1, n = list.length; k < n; i += 1,
 k += 1)
    list[i] = list[k];
  list.pop();
}
```

性能是原生调用的1.5倍。

事件触发

在事件触发时，监听器拥有的参数数量是任意的。

```
EventEmitter.prototype.emit = function emit(type) {
  var er, handler, len, args, i, events, domain;
  var needDomainExit = false;
  var doError = (type === 'error');

  events = this._events;
  ...

  handler = events[type];

  if (!handler)
    return false;
  ...
  var isFn = typeof handler === 'function';
  len = arguments.length;
  switch (len) {
    // fast cases
```

```
    case 1:
      emitNone(handler, isFn, this);
      break;
    case 2:
      emitOne(handler, isFn, this, arguments[1]);
      break;
    case 3:
      emitTwo(handler, isFn, this, arguments[1], arguments[2]);
      break;
    case 4:
      emitThree(handler, isFn, this, arguments[1], arguments[2], arguments[3]);
      break;
    // slower
    default:
      args = new Array(len - 1);
      for (i = 1; i < len; i++)
        args[i - 1] = arguments[i];
      emitMany(handler, isFn, this, args);
  }
  ...
  return true;
```

把不定参数的函数调用转变成固定参数的函数调用，且最多支持到三个参数。超过3个参数则调用 `emitMany` . 结果不言而喻，我们还是比较下会差多少，以三个参数为例： jsperf 显示的性能差距在1倍左右。

> 深入了解 https://github.com/iojs/io.js/pull/601

event在node中的应用

监控文件变化，通知感兴趣的观察者。

```
1389 function FSWatcher() {
1390   EventEmitter.call(this);
1391
1392   var self = this;
1393   this._handle = new FSEvent();
1394   this._handle.owner = this;
1395
1396   this._handle.onchange = function(status, event, filename) {
1397     if (status < 0) {
1398       self._handle.close();
1399       const error = !filename ?
1400           errnoException(status, 'Error watching file for changes:') :
1401           errnoException(status,
1402                          `Error watching file ${filename} for changes:`);
1403       error.filename = filename;
1404       self.emit('error', error);
1405     } else {
1406       self.emit('change', event, filename);
1407     }
1408   };
1409 }
1410 util.inherits(FSWatcher, EventEmitter);
```

L1410, FSWatcher 对象继承 EventEmitter，使自身有了EventEmitter的方法。L1404， 当底层发生错误时，会发出通知事件 `error` 。L1406， 文件发生变化时，FSWatcher 对象发射 `change` 事件，具体的变化由 *event*标识，*filename*标识文件名。

L1396, 挂在 `FSEvent` 对象上的方法 `onchange` 作为 C++调用 Javascript 的回调，在不同的平台实现方式也不一样， 我们在文件系统章节将详细讲述。

上述是 fs 模块监听文件变化的实现，并导出API: `fs.watch()` 给外部使用，另外还有一个 `fs.watchFile()` 。 我们查看官方文档：

> fs.watchFile(filename, [options], listener)
>
> Stability: 2 - Unstable. Use fs.watch instead, if available.
>
> Watch for changes on filename.
>
> fs.watch(filename, [options], [listener])
>
> Stability: 2 - Unstable. Not available on all platforms.

- fs.watch() 官方建议使用。
- fs.watch() 并不是全平台支持，只有 OSX 和 Windows 支持recursive选项。
- fs.watch() 监听文件或目录， fs.watchFile() 监听文件。

fs.watch() 如果传入 listener, 如下：

```
fs.watch('somedir', function (event, filename) {
  console.log('event is: ' + event);
  if (filename) {
    console.log('filename provided: ' + filename);
  }
});
```

则默认添加函数 callback 到 `change` 事件的观察者中。当然也可以换个姿势，如：

```
var watcher = fs.watch('somedir');
watcher.on('change', function (event, filename) {
  console.log('event is: ' + event);
  if (filename) {
    console.log('filename provided: ' + filename);
  }
}).on('error', function(error) {

})
```

可以实现链式调用，比如符合目前很火的Reactive Programming。 RP编程范式提高了编码的抽象程度，你可以更好地关注在商业逻辑中各种事件的联系避免大量细节而琐碎的实现，使得编码更加简洁。

逐行读取 (Readline)

我们来看看逐行读取对键盘输入的处理， 这涉及到比较复杂的状态机和事件发送，是学习事件模块非常好的一个例子。

```
Interface.prototype._onLine = function(line) {
  if (this._questionCallback) {
    var cb = this._questionCallback;
    this._questionCallback = null;
    this.setPrompt(this._oldPrompt);
    cb(line);
  } else {
    this.emit('line', line);
  }
};
```

如果没有预先设定指定的query，然后用户应答后触发指定的callback，那么 `Interface` 对象会触发 `line` 事件。 在 input 流接受了一个 `\n` 时触发，通常在用户敲击回车或者返回时接收。 这是一个监听用户输入的利器。 监听 line 事件的示例:

```
var readline = require('readline');
var rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});
rl.on('line', function (cmd) {
  console.log('You just typed: '+ cmd);
});
```

该模块对复合功能按键，比如 Ctrl + c, Ctrl + z也做了相应的处理, 我们拿对 Ctrl + c 的代码进行分析：

```
678 Interface.prototype._ttyWrite = function(s, key) {
679   key = key || {};
680
681   // Ignore escape key - Fixes #2876
682   if (key.name == 'escape') return;
683
684   if (key.ctrl && key.shift) {
685     /* Control and shift pressed */
686     switch (key.name) {
687       case 'backspace':
688         this._deleteLineLeft();
689         break;
690
691       case 'delete':
692         this._deleteLineRight();
693         break;
694     }
695
696   } else if (key.ctrl) {
697     /* Control key pressed */
698
699     switch (key.name) {
700       case 'c':
701         if (this.listenerCount('SIGINT') > 0) {
702           this.emit('SIGINT');
703         } else {
704           // This readline instance is finished
705           this.close();
706         }
707         break;
708     省略...
709 }
```

- L681-L682, 忽略 `ESC` 键。
- L684, 首先判断是否是 Ctrl 和 Shift复合键同时按下，如果是则L685-L694优先处理。
- L696, 如果是按下 Ctrl 键，L699 继续判断，如果另一个是 `c` ，默认是关闭对象。

- L701, 如果外部有观察者, 则发送 `SIGINT` 事件，交由观察者处理。

REPL

一个 Read-Eval-Print-Loop（REPL，读取-执行-输出循环）既可用于独立程序也可很容易地被集成到其它程序中。REPL 提供了一种交互地执行 JavaScript 并查看输出的方式。它可以被用作调试、测试或仅仅尝试某些东西。

在命令行中不带任何参数执行 node 您便会进入 REPL。它提供了一个简单的 Emacs 行编辑。

REPLServer 继承 Interface，如代码所示： `inherits(REPLServer, rl.Interface);`

并监听 line 事件, 自定义关键字，以支持交互式的命令。

```
$ NODE_DEBUG=REPL node
REPL 37391: line ".help"
break Sometimes you get stuck, this gets you out
clear Alias for .break
exit  Exit the repl
help  Show repl options
load  Load JS from a file into the REPL session
save  Save all evaluated commands in this REPL session to a file
```

我们看下代码实现：

```
399   self.on('line', function(cmd) {
 400     debug('line %j', cmd);
 401     sawSIGINT = false;
 402
 403     // leading whitespaces in template literals should not be trimmed.
 404     if (self._inTemplateLiteral) {
 405       self._inTemplateLiteral = false;
 406     } else {
 407       cmd = self.lineParser.parseLine(cmd);
 408     }
 409
 410     // Check to see if a REPL keyword was used. If it returns true,
 411     // display next prompt and return.
 412     if (cmd && cmd.charAt(0) === '.' && isNaN(parseFloat(cmd))) {
 413       var matches = cmd.match(/^\.([^\s]+)\s*(.*)$/);
 414       var keyword = matches && matches[1];
 415       var rest = matches && matches[2];
 416       if (self.parseREPLKeyword(keyword, rest) === true) {
 417         return;
 418       } else if (!self.bufferedCommand) {
 419         self.outputStream.write('Invalid REPL keyword\n');
 420         finish(null);
 421         return;
 422       }
 423     }
 424     ...
 425   }
```

- L400， 通过设置环境变量NODE_DEBUG=REPL打开调试功能。
- L407， 解析 cmd 输入， 处理正则的情况。
- L412, 查看是否以 `.` 开头，并且不是浮点数，则利用正则匹配字符串，
  - 以 .help 为例，得到的 `matches` 为 `[ '.help', 'help', '', index: 0, input: '.help' ]` ， keyword 为 help, rest 为 ''.
- L416, 通过 keyword 从 commands 对象找到对应的方法执行。

REPL实例

一个在curl(1)上运行的REPL实例的例子可以查看这里：
https://gist.github.com/2053342

EventEmitter vs Callbacks

- EventEmitter
  - 可以通知多个listeners
  - 一般被调用多次。
- Callback
  - 最多通知一个listener
  - 通常被调用一次，无论操作是成功还是失败。

总结

Event 模块是观察者设计模式的典型应用。同时也是Reactive Programming的精髓所在。

参考

[1].https://segmentfault.com/a/1190000005051034

异常处理与domain

异步异常捕获

由于node的回调异步特性，无法通过try catch来捕捉所有的异常：

```
try {
  process.nextTick(function () {
    foo.bar();
  });
} catch (err) {
  //can not catch it
}
```

如果try catch能够捕获所有的异常，这样我们可以在代码出现一些非预期的错误时，能够记录下错误的同时，友好的给调用者返回一个500错误。可惜，try catch无法捕获异步中的异常。所以我们能做的只能是：

```
app.get('/index', function (req, res) {
  // 业务逻辑
});

process.on('uncaughtException', function (err) {
  logger.error(err);
});
```

这个时候，虽然我们可以记录下这个错误的日志，且进程也不会异常退出，但是我们是没有办法对发现错误的请求友好返回的，因为异常处理只返回给我们一个冷冰冰的 `error`，脱离了上下文，我们只能够让它超时返回。

domain

在node v0.8+版本的时候，发布了一个模块domain。这个模块做的就是try catch所无法做到的：捕捉异步回调中出现的异常。 于是乎，我们上面那个无奈的例子好像有了解决的方案：

```
var domain = require('domain');

//引入一个domain的中间件，将每一个请求都包裹在一个独立的domain中
//domain来处理异常
app.use(function (req,res, next) {
  var d = domain.create();
  //监听domain的错误事件
  d.on('error', function (err) {
    logger.error(err);
    res.statusCode = 500;
    res.json({sucess:false, messag: '服务器异常'});
    d.dispose();
  });

  d.add(req);
  d.add(res);
  d.run(next);
});
```

`domain` 虽然捕捉到了异常，但是还是由于异常而导致的堆栈丢失会导致内存泄漏，所以出现这种情况的时候还是需要重启这个进程的。

Domain剖析

Domain 自身其实也是 Event 模块一个典型的应用， 它通过事件的方式来传递捕获的错误。

```
inherits(Domain, EventEmitter);

function Domain() {
  EventEmitter.call(this);

  this.members = [];
}
```

另外， domain 为了支持深层次的嵌套， 提供了 `Domain#enter` 和 `Domain#exit` 的 API。 先来看 `enter` 的实现，

```
Domain.prototype.enter = function() {
  if (this._disposed) return;

  // note that this might be a no-op, but we still need
  // to push it onto the stack so that we can pop it later.
  exports.active = process.domain = this;
  stack.push(this);
  _domain_flag[0] = stack.length;
};
```

设置当前活跃的 `domain` , 并且为了便于回溯，将当前的 `domain` 加入到队列的后面，更新栈的深度。

再看 `exit` 实现，

```
Domain.prototype.exit = function() {
  // skip disposed domains, as usual, but also don't do anything
 if this
  // domain is not on the stack.
  var index = stack.lastIndexOf(this);
  if (this._disposed || index === -1) return;

  // exit all domains until this one.
  stack.splice(index);
  _domain_flag[0] = stack.length;

  exports.active = stack[stack.length - 1];
  process.domain = exports.active;
};
```

相反的， 退出当前的 `domain` , 更新长度，设置当前活跃的 `domain` 。

读者可能好奇，我并没有显式地调用 `enter` , `exit` , 而只是简单的创建了一个domain, 怎么会达到这种效果？

读者可以看看 `AsyncWrap::MakeCallback()` , 每次C++ --> JS, 都会检查domain, 如果使用，则会显式地调用他们。 其他地方读者可以自行寻找。

为了解决不在当前作用域的异常处理，Domain 也提供 `Domain#add` 和 `Domain#remove` 来增加 `emitter` 或者 `Timer` 。

回到事件的根本，什么时候触发domain的error事件？

```
process._fatalException = function(er) {
      var caught;

      if (process.domain && process.domain._errorHandler)
        caught = process.domain._errorHandler(er) || caught;

      if (!caught)
        caught = process.emit('uncaughtException', er);

      // If someone handled it, then great.  otherwise, die in C
++ land
      // since that means that we'll exit the process, emit the
'exit' event
      if (!caught) {
        try {
          if (!process._exiting) {
            process._exiting = true;
            process.emit('exit', 1);
          }
        } catch (er) {
          // nothing to be done about it at this point.
        }

      // if we handled an error, then make sure any ticks get pr
ocessed
      } else {
        NativeModule.require('timers').setImmediate(process._tic
kCallback);
      }

      return caught;
    };
```

如果当前 process 使用了 domain, 也是就 `process.domain` 不为空，就调用 `_errorHandler` 来处理， 当前也存在没有处理的情况，职责链来到 process，process 则触发 `uncaughtException` 事件。

总结

domain很强大，但它只能捕获在其作用域范围内的异常。对于非预期的异常产生的时候， 我们最好让当前请求超时，然后让这个进程停止服务，之后重新启动。

但始终 Domain 在异常处理上有各种不完美，目前该模块处于即将废除阶段，取代他的可能是另一种机制。

详细讨论见：

> https://github.com/nodejs/node/issues/66

参考

- http://node.alibaba-inc.com/post/async-error-handle-and-domain.html?spm=0.0.0.0.7r8vQ2

Stream 流

从早先的unix开始，stream便开始进入了人们的视野，在过去的几十年的时间里，它被证明是一种可依赖的编程方式，它可以将一个大型的系统拆成一些很小的部分，并且让这些部分之间完美地进行合作。在unix中，我们可以使用 `|` 符号来实现流。在node中，node内置的stream模块已经被多个核心模块使用，同时也可以被用户自定义的模块使用。和unix类似，node中的流模块的基本操作符叫做 `.pipe()` ，同时你也可以使用一个后压机制来应对那些对数据消耗较慢的对象。

为什么应该使用流

在node中，I/O都是异步的，所以在和硬盘以及网络的交互过程中会涉及到传递回调函数的过程。你之前可能会写出这样的代码：

```
    var http = require('http');
    var fs = require('fs');

    var server = http.createServer(function (req, res) {
        fs.readFile(__dirname + 'data.txt', function (err, data)
{
            res.end(data);
        });
    });
    server.listen(8000);
```

上面的这段代码并没有什么问题，但是在每次请求时，我们都会把整个 `data.txt` 文件读入到内存中，然后再把结果返回给客户端。想想看，如果 `data.txt` 文件非常大，在响应大量用户的并发请求时，程序可能会消耗大量的内存，这样很可能会造成用户连接缓慢的问题。

其次，上面的代码可能会造成很不好的用户体验，因为用户在接收到任何的内容之前首先需要等待程序将文件内容完全读入到内存中。

所幸的是，`(req,res)`参数都是流对象，这意味着我们可以使用一种更好的方法来实现上面的需求：

```
var http = require('http');
var fs = require('fs');

var server = http.createServer(function (req, res) {
    var stream = fs.createReadStream(__dirname + '/data.txt'
);
    stream.pipe(res);
});
server.listen(8000);
```

在这里，`.pipe()`方法会自动帮助我们监听`data`和`end`事件。上面的这段代码不仅简洁，而且`data.txt`文件中每一小段数据都将源源不断的发送到客户端。

除此之外，使用`.pipe()`方法还有别的好处，比如说它可以自动控制后端压力，以便在客户端连接缓慢的时候node可以将尽可能少的缓存放到内存中。

想要将数据进行压缩？我们可以使用相应的流模块完成这项工作!

```
var http = require('http');
var fs = require('fs');
var oppressor = require('oppressor');

var server = http.createServer(function (req, res) {
    var stream = fs.createReadStream(__dirname + '/data.txt'
);
    stream.pipe(oppressor(req)).pipe(res);
});
server.listen(8000);
```

通过上面的代码，我们成功的将发送到浏览器端的数据进行了gzip压缩。我们只是使用了一个oppressor模块来处理这件事情。

一旦你学会使用流api，你可以将这些流模块像搭乐高积木或者像连接水管一样拼凑起来，从此以后你可能再也不会去使用那些没有流API的模块获取和推送数据了。

流模块基础

nodejs 底层一共提供了4个流， Readable 流、Writable 流、Duplex 流和 Transform 流。

使用情景	类	需要重写的方法
只读	Readable	_read
只写	Writable	_write
双工	Duplex	_read, _write
操作被写入数据，然后读出结果	Transform	_transform, _flush

pipe

无论哪一种流，都会使用 `.pipe()` 方法来实现输入和输出。

`.pipe()` 函数很简单，它仅仅是接受一个源头 `src` 并将数据输出到一个可写的流 `dst` 中：

```
src.pipe(dst)
```

`.pipe(dst)` 将会返回 `dst` 因此你可以链式调用多个流:

```
a.pipe(b).pipe(c).pipe(d)
```

上面的代码也可以等价为：

```
a.pipe(b);
b.pipe(c);
c.pipe(d);
```

这和你在unix中编写流代码很类似：

```
a | b | c | d
```

只不过此时你是在node中编写而不是在shell中！

readable流

Readable流可以产出数据，你可以将这些数据传送到一个writable，transform或者duplex流中，只需要调用 `pipe()` 方法:

```
readableStream.pipe(dst)
```

创建一个readable流

现在我们就来创建一个readable流！

```
var Readable = require('stream').Readable;

var rs = new Readable;
rs.push('beep ');
rs.push('boop\n');
rs.push(null);

rs.pipe(process.stdout);
```

下面运行代码：

```
$ node read0.js
beep boop
```

在上面的代码中 `rs.push(null)` 的作用是告诉 `rs` 输出数据应该结束了。

需要注意的一点是我们在将数据输出到 `process.stdout` 之前已经将内容推送进readable流 `rs` 中，但是所有的数据依然是可写的。

这是因为在你使用 `.push()` 将数据推进一个readable流中时，一直要到另一个东西来消耗数据之前，数据都会存在一个缓存中。

然而，在更多的情况下，我们想要的是当需要数据时数据才会产生，以此来避免大量的缓存数据。

我们可以通过定义一个 `._read` 函数来实现按需推送数据:

```
var Readable = require('stream').Readable;
var rs = Readable();

var c = 97;
rs._read = function () {
    rs.push(String.fromCharCode(c++));
    if (c > 'z'.charCodeAt(0)) rs.push(null);
};

rs.pipe(process.stdout);
```

代码的运行结果如下所示:

```
$ node read1.js
abcdefghijklmnopqrstuvwxyz
```

在这里我们将字母 `a` 到 `z` 推进了rs中，但是只有当数据消耗者出现时，数据才会真正实现推送。

`_read` 函数也可以获取一个 `size` 参数来指明消耗者想要读取多少比特的数据，但是这个参数是可选的。

需要注意到的是你可以使用 `util.inherit()` 来继承一个Readable流。

为了说明只有在数据消耗者出现时， `_read` 函数才会被调用，我们可以将上面的代码简单的修改一下：

```
var Readable = require('stream').Readable;
var rs = Readable();

var c = 97 - 1;

rs._read = function () {
    if (c >= 'z'.charCodeAt(0)) return rs.push(null);

    setTimeout(function () {
        rs.push(String.fromCharCode(++c));
    }, 100);
};

rs.pipe(process.stdout);

process.on('exit', function () {
    console.error('\n_read() called ' + (c - 97) + ' times');
});
process.stdout.on('error', process.exit);
```

运行上面的代码我们可以发现如果我们只请求5比特的数据，那么 `_read` 只会运行5次：

```
$ node read2.js | head -c5
abcde
_read() called 5 times
```

在上面的代码中， `setTimeout` 很重要，因为操作系统需要花费一些时间来发送程序结束信号。

另外, `process.stdout.on('error',fn)` 处理器也很重要，因为当 `head` 不再关心我们的程序输出时，操作系统将会向我们的进程发送一个 `SIGPIPE` 信号，此时 `process.stdout` 将会捕获到一个 `EPIPE` 错误。

上面这些复杂的部分在和操作系统相关的交互中是必要的，但是如果你直接和node中的流交互的话，则可有可无。

如果你创建了一个readable流，并且想要将任何的值推送到其中的话，确保你在创建流的时候指定了objectMode参数, `Readable({ objectMode: true })` 。

消耗一个readable流

大部分时候，将一个readable流直接pipe到另一种类型的流或者使用through或者concat-stream创建的流中，是一件很容易的事情。但是有时我们也会需要直接来消耗一个readable流。

```
process.stdin.on('readable', function () {
    var buf = process.stdin.read();
    console.dir(buf);
});
```

代码运行结果如下所示：

```
$ (echo abc; sleep 1; echo def; sleep 1; echo ghi) | node consum
e0.js
<Buffer 61 62 63 0a>
<Buffer 64 65 66 0a>
<Buffer 67 68 69 0a>
null
```

当数据可用时， `readable` 事件将会被触发，此时你可以调用 `.read()` 方法来从缓存中获取这些数据。

当流结束时， `.read()` 将返回 `null` ，因为此时已经没有更多的字节可以供我们获取了。

你也可以告诉 `.read()` 方法来返回 `n` 个字节的数据。虽然所有核心对象中的流都支持这种方式，但是对于对象流来说这种方法并不可用。

下面是一个例子，在这里我们制定每次读取3个字节的数据：

```
process.stdin.on('readable', function () {
    var buf = process.stdin.read(3);
    console.dir(buf);
});
```

运行上面的例子，我们将获取到不完整的数据:

```
$ (echo abc; sleep 1; echo def; sleep 1; echo ghi) | node consum
e1.js
<Buffer 61 62 63>
<Buffer 0a 64 65>
<Buffer 66 0a 67>
```

这是因为多余的数据都留在了内部的缓存中，因此这个时候我们需要告诉node我们还对剩下的数据感兴趣，我们可以使用 `.read(0)` 来完成这件事：

```
process.stdin.on('readable', function () {
    var buf = process.stdin.read(3);
    console.dir(buf);
    process.stdin.read(0);
});
```

到现在为止我们的代码和我们所期望的一样了！

```
$ (echo abc; sleep 1; echo def; sleep 1; echo ghi) | node consum
e2.js
<Buffer 61 62 63>
<Buffer 0a 64 65>
<Buffer 66 0a 67>
<Buffer 68 69 0a>
```

我们也可以使用 `.unshift()` 方法来放置多余的数据。

使用 `unshift()` 方法能够放置我们进行不必要的缓存拷贝。在下面的代码中我们将创建一个分割新行的可读解析器:

```
var offset = 0;

process.stdin.on('readable', function () {
    var buf = process.stdin.read();
    if (!buf) return;
    for (; offset < buf.length; offset++) {
        if (buf[offset] === 0x0a) {
            console.dir(buf.slice(0, offset).toString());
            buf = buf.slice(offset + 1);
            offset = 0;
            process.stdin.unshift(buf);
            return;
        }
    }
    process.stdin.unshift(buf);
});
```

代码的运行结果如下所示：

```
$ tail -n +50000 /usr/share/dict/american-english | head -n10 |
node lines.js
'hearties'
'heartiest'
'heartily'
'heartiness'
'heartiness\'s'
'heartland'
'heartland\'s'
'heartlands'
'heartless'
'heartlessly'
```

当然，已经有很多这样的模块比如split来帮助你完成这件事情，你完全不需要自己写一个。

writable流

一个writable流指的是只能流进不能流出的流:

```
src.pipe(writableStream)
```

创建一个writable流

只需要定义一个 `._write(chunk,enc,next)` 函数，你就可以将一个readable流的数据释放到其中：

```
var Writable = require('stream').Writable;
var ws = Writable();
ws._write = function (chunk, enc, next) {
    console.dir(chunk);
    next();
};

process.stdin.pipe(ws);
```

代码运行结果如下所示：

```
$ (echo beep; sleep 1; echo boop) | node write0.js
<Buffer 62 65 65 70 0a>
<Buffer 62 6f 6f 70 0a>
```

第一个参数， `chunk` 代表写进来的数据。

第二个参数 `enc` 代表编码的字符串，但是只有在 `opts.decodeString` 为 `false` 的时候你才可以写一个字符串。

第三个参数， `next(err)` 是一个回调函数，使用这个回调函数你可以告诉数据消耗者可以写更多的数据。你可以有选择性的传递一个错误对象 `error` ，这时会在流实体上触发一个 `emit` 事件。

在从一个readable流向一个writable流传数据的过程中，数据会自动被转换为 `Buffer` 对象，除非你在创建writable流的时候制定了 `decodeStrings` 参数为 `false` , `Writable({decodeStrings: false})` 。

如果你需要传递对象，需要指定 `objectMode` 参数为 `true` ，`Writable({ objectMode: true })` 。

向一个writable流中写东西

如果你需要向一个writable流中写东西，只需要调用 `.write(data)` 即可。

```
process.stdout.write('beep boop\n');
```

为了告诉一个writable流你已经写完毕了，只需要调用 `.end()` 方法。你也可以使用 `.end(data)` 在结束前再写一些数据。

```
var fs = require('fs');
var ws = fs.createWriteStream('message.txt');

ws.write('beep ');

setTimeout(function () {
    ws.end('boop\n');
}, 1000);
```

运行结果如下所示:

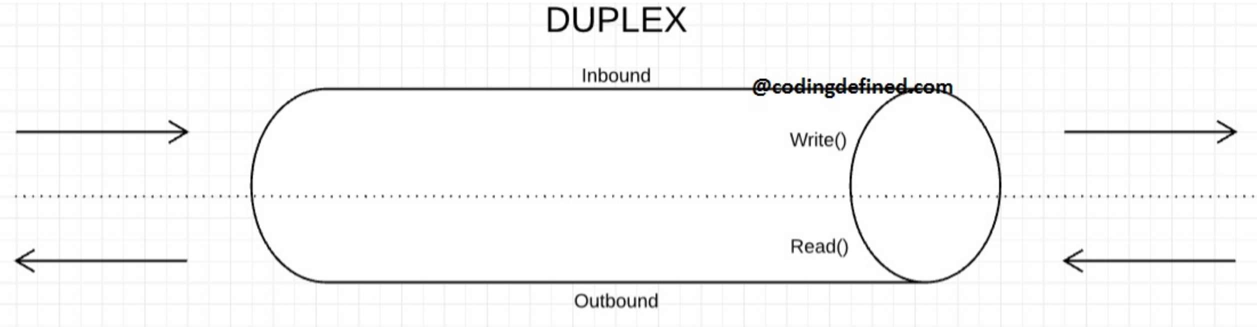
```
$ node writing1.js
$ cat message.txt
beep boop
```

如果你在创建writable流时指定了 `highWaterMark` 参数，那么当没有更多数据写入时，调用 `.write()` 方法将会返回false。

如果你想要等待缓存情况，可以监听 `drain` 事件。

duplex流

Duplex流是一个可读也可写的流，全双工。 如图：



代码实现上：

```
const Readable = require('_stream_readable');
const Writable = require('_stream_writable');

util.inherits(Duplex, Readable);

var keys = Object.keys(Writable.prototype);
for (var v = 0; v < keys.length; v++) {
  var method = keys[v];
  if (!Duplex.prototype[method])
    Duplex.prototype[method] = Writable.prototype[method];
}
```

`Duplex` 首先继承了 `Readable` , 因为 javascript 没有 C++的多重继承的特性，所以 遍历 `Writable` 的原型方法然后赋值到 `Duplex` 的原型上。

transform流

转换流（Transform streams）是一种输出由输入计算所得的双工流。它同时实现了 Readable 和 Writable 接口。

Node中的转换流有：

- zlib streams
- crypto streams

你可以将transform流想象成一个流的中间部分，它可以读也可写，但是并不保存数据，它只负责处理流经它的数据。

总结

流式处理的优势: 将功能切分，并通过管道组合。

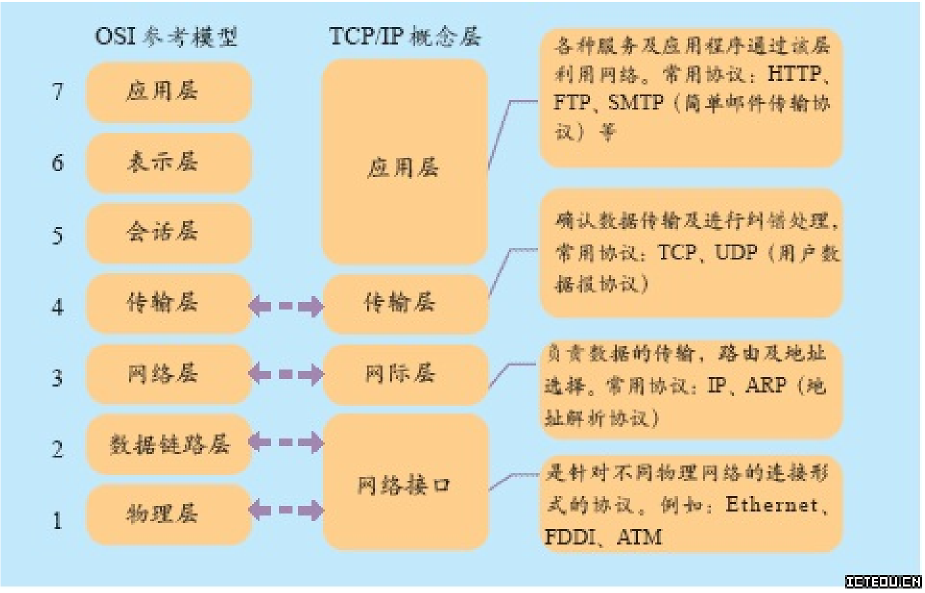
参考

https://github.com/substack/stream-handbook

网络 (Net)

网络模型

ISO制定的OSI参考模型的过于庞大、复杂招致了许多批评。与此对照，由技术人员自己开发的TCP/ IP协议栈获得了更为广泛的应用。如图所示，是TCP/IP参考模型和OSI参考模型的对比示意图。



UDP vs TCP

- TCP(Transmission Control Protocol)：传输控制协议
- UDP(User Datagram Protocol)：用户数据报协议

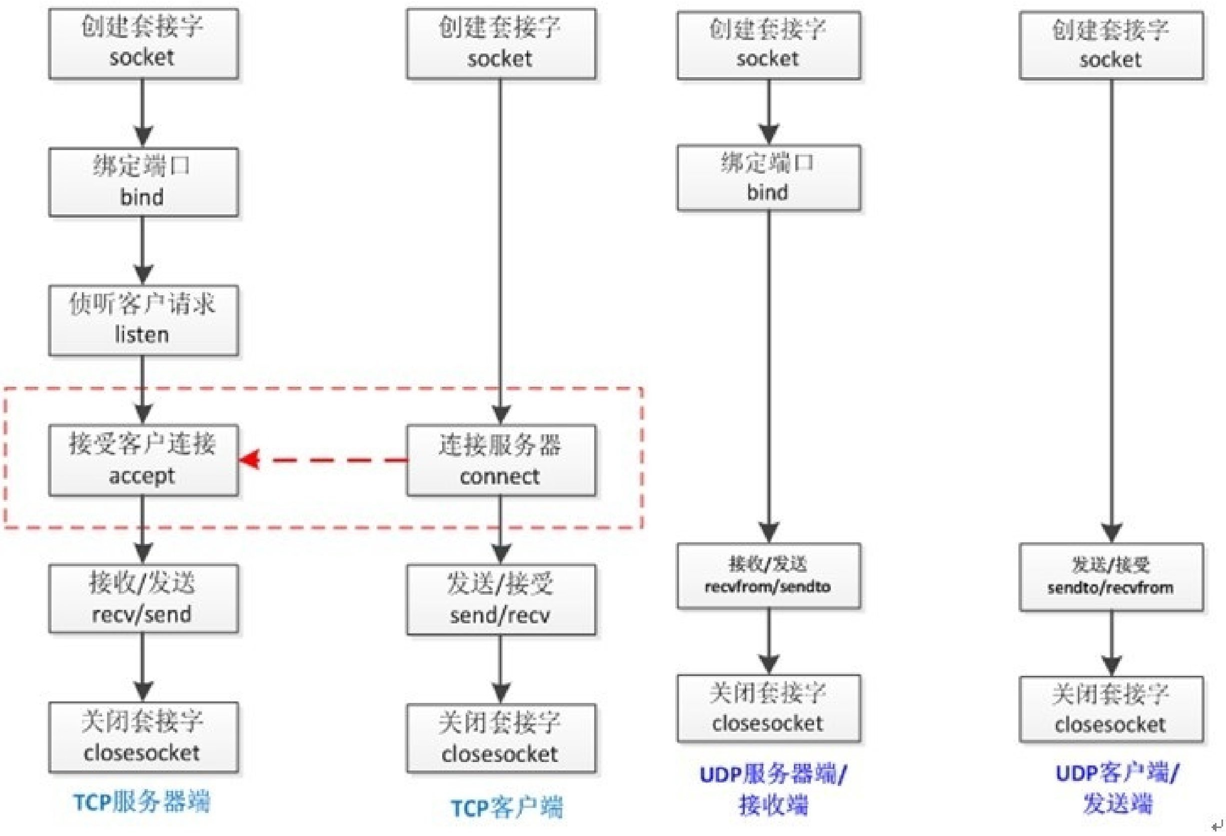
主要在于连接性(Connectivity)、可靠性(Reliability)、有序性(Ordering)、有界性(Boundary)、拥塞控制(Congestion or Flow control)、传输速度(Speed)、量级(Heavy/Light weight)、头部大小(Header size)等差异。

主要差异：

- TCP是面向连接(Connection oriented)的协议，UDP是无连接(Connection less)协议；
    - TCP用三次握手建立连接：1) Client向server发送SYN；2) Server接收到SYN，回复Client一个SYN-ACK；3)Client接收到SYN_ACK，回复Server一个ACK。到此，连接建成。UDP发送数据前不需要建立连接。

- TCP可靠，UDP不可靠；
    - TCP丢包会自动重传，UDP不会。
- TCP有序，UDP无序；
    - 消息在传输过程中可能会乱序，后发送的消息可能会先到达，TCP会对其进行重排序，UDP不会。

从程序实现的角度来看，可以用下图来进行描述。



从上图也能清晰的看出，TCP通信需要服务器端侦听listen、接收客户端连接请求accept，等待客户端connect建立连接后才能进行数据包的收发（recv/send）工作。而UDP则服务器和客户端的概念不明显，服务器端即接收端需要绑定端口，等待客户端的数据的到来。后续便可以进行数据的收发（recvfrom/sendto）工作。

Socket 抽象

Socket 是对 TCP/IP 协议族的一种封装，是应用层与TCP/IP协议族通信的中间软件抽象层。它把复杂的TCP/IP协议族隐藏在Socket接口后面，对用户来说，一组简单的接口就是全部，让Socket去组织数据，以符合指定的协议。

Socket 还可以认为是一种网络间不同计算机上的进程通信的一种方法，利用三元组（ip地址，协议，端口）就可以唯一标识网络中的进程，网络中的进程通信可以利用这个标志与其它进程进行交互。

Socket 起源于 Unix ，Unix/Linux 基本哲学之一就是“一切皆文件”，都可以用“打开(open) –> 读写(write/read) –> 关闭(close)”模式来进行操作。因此 Socket 也被处理为一种特殊的文件。

C++层绑定

TCP的绑定导出：

```
void TCPWrap::Initialize(Local<Object> target,
                         Local<Value> unused,
                         Local<Context> context) {
  Environment* env = Environment::GetCurrent(context);

  Local<FunctionTemplate> t = env->NewFunctionTemplate(New);
  t->SetClassName(FIXED_ONE_BYTE_STRING(env->isolate(), "TCP"));
  t->InstanceTemplate()->SetInternalFieldCount(1);

  // Init properties
  t->InstanceTemplate()->Set(String::NewFromUtf8(env->isolate(),
"reading"),
                             Boolean::New(env->isolate(), false)
);
  t->InstanceTemplate()->Set(String::NewFromUtf8(env->isolate(),
"owner"),
                             Null(env->isolate()));
  t->InstanceTemplate()->Set(String::NewFromUtf8(env->isolate(),
"onread"),
                             Null(env->isolate()));
  t->InstanceTemplate()->Set(String::NewFromUtf8(env->isolate(),
                                                 "onconnection")
,
                             Null(env->isolate()));


  env->SetProtoMethod(t, "close", HandleWrap::Close);
```

```
  env->SetProtoMethod(t, "ref", HandleWrap::Ref);
  env->SetProtoMethod(t, "unref", HandleWrap::Unref);

  StreamWrap::AddMethods(env, t, StreamBase::kFlagHasWritev);

  env->SetProtoMethod(t, "open", Open);
  env->SetProtoMethod(t, "bind", Bind);
  env->SetProtoMethod(t, "listen", Listen);
  env->SetProtoMethod(t, "connect", Connect);
  env->SetProtoMethod(t, "bind6", Bind6);
  env->SetProtoMethod(t, "connect6", Connect6);
  env->SetProtoMethod(t, "getsockname",
                      GetSockOrPeerName<TCPWrap, uv_tcp_getsockn
ame>);
  env->SetProtoMethod(t, "getpeername",
                      GetSockOrPeerName<TCPWrap, uv_tcp_getpeern
ame>);
  env->SetProtoMethod(t, "setNoDelay", SetNoDelay);
  env->SetProtoMethod(t, "setKeepAlive", SetKeepAlive);

#ifdef _WIN32
  env->SetProtoMethod(t, "setSimultaneousAccepts", SetSimultaneo
usAccepts);
#endif

  target->Set(FIXED_ONE_BYTE_STRING(env->isolate(), "TCP"), t->G
etFunction());
  env->set_tcp_constructor_template(t);

  // Create FunctionTemplate for TCPConnectWrap.
  Local<FunctionTemplate> cwt =
      FunctionTemplate::New(env->isolate(), NewTCPConnectWrap);
  cwt->InstanceTemplate()->SetInternalFieldCount(1);
  cwt->SetClassName(FIXED_ONE_BYTE_STRING(env->isolate(), "TCPCo
nnectWrap"));
  target->Set(FIXED_ONE_BYTE_STRING(env->isolate(), "TCPConnectW
rap"),
              cwt->GetFunction());
}
```

`TCPWrap` 导出了 TCP 类，TCPConnectWrap 类，并且我们看到对 IPV6协议族的支持：`bind6`，`connect6`。

TCP Socket

Node.js 的 Net模块也对 TCP socket 进行了抽象封装：

```
function Socket(options) {
  if (!(this instanceof Socket)) return new Socket(options);

  this._connecting = false;
  this._hadError = false;
  this._handle = null;
  this._parent = null;
  this._host = null;

  if (typeof options === 'number')
    options = { fd: options }; // Legacy interface.
  else if (options === undefined)
    options = {};

  stream.Duplex.call(this, options);

  if (options.handle) {
    this._handle = options.handle; // private
  } else if (options.fd !== undefined) {
    this._handle = createHandle(options.fd);
    this._handle.open(options.fd);
    if ((options.fd == 1 || options.fd == 2) &&
        (this._handle instanceof Pipe) &&
        process.platform === 'win32') {
      // Make stdout and stderr blocking on Windows
      var err = this._handle.setBlocking(true);
      if (err)
        throw errnoException(err, 'setBlocking');
    }
    this.readable = options.readable !== false;
    this.writable = options.writable !== false;
  } else {
    // these will be set once there is a connection
```

```
    this.readable = this.writable = false;
  }

  // shut down the socket when we're finished with it.
  this.on('finish', onSocketFinish);
  this.on('_socketEnd', onSocketEnd);

  initSocketHandle(this);

  // ...
}
util.inherits(Socket, stream.Duplex);
```

首先 `Socket` 是一个全双工的 Stream，所以继承了 Duplex。通过 `createHandle` 创建套接字并赋值到 `this._handle` 上。

同时监听 `finish` , `_socketEnd` 事件，

粘包

> 一般所谓的TCP粘包是在一次接收数据不能完全地体现一个完整的消息数据。TCP通讯为何存在粘包呢？主要原因是TCP是以流的方式来处理数据，再加上网络上MTU的往往小于在应用处理的消息数据，所以就会引发一次接收的数据无法满足消息的需要，导致粘包的存在。处理粘包的唯一方法就是制定应用层的数据通讯协议，通过协议来规范现有接收的数据是否满足消息数据的需要。

情况分析

TCP粘包通常在流传输中出现，UDP则不会出现粘包，因为UDP有消息边界，发送数据段需要等待缓冲区满了才将数据发送出去，当满的时候有可能不是一条消息而是几条消息合并在换中去内，在成粘包；另外接收数据端没能及时接收缓冲区的包，造成了缓冲区多包合并接收，也是粘包。

解决办法

- 自定义应用层协议；
- 不使用Nagle算法, 使用提供的 API： `socket.setNoDelay` 。

UDP

组播

- https://en.wikipedia.org/wiki/Multicast#IP_multicast%EF%BC%89%EF%BC%8C%E5%85%B7

UDP Socket

```
function Socket(type, listener) {
  EventEmitter.call(this);

  if (typeof type === 'object') {
    var options = type;
    type = options.type;
  }

  var handle = newHandle(type);
  handle.owner = this;

  this._handle = handle;
  this._receiving = false;
  this._bindState = BIND_STATE_UNBOUND;
  this.type = type;
  this.fd = null; // compatibility hack

  // If true - UV_UDP_REUSEADDR flag will be set
  this._reuseAddr = options && options.reuseAddr;

  if (typeof listener === 'function')
    this.on('message', listener);
}
util.inherits(Socket, EventEmitter);
```

UDP 继承了 `EventEmitter` , 同样也支持 IPV4和 IPV6协议， 由 `type` 区分， `this._reuseAddr` 标识是否要使用选项： `SO_REUSEADDR` 。

SO_REUSEADDR允许完全重复的捆绑：当一个IP地址和端口绑定到某个套接口上时，还允许此IP地址和端口捆绑到另一个套接口上。一般来说，这个特性仅在支持多播的系统上才有，而且只对UDP套接口而言（TCP不支持多播）。

总结

从笔者的经验看，尽量不要尝试去使用 `UDP` ，除非你知道丢包了对于应用是没有影响的，否则排查网络丢包会使人崩溃的！

参考

- https://en.wikipedia.org/wiki/Nagle's_algorithm

应用构建

创建TCP服务端

下面是一个在NodeJS中创建TCP服务端套接字的简单例子，相关说明见代码注释。

```
var net = require('net');

var HOST = '127.0.0.1';
var PORT = 6969;

// 创建一个TCP服务器实例，调用listen函数开始监听指定端口
// 传入net.createServer()的回调函数将作为”connection“事件的处理函数
// 在每一个“connection”事件中，该回调函数接收到的socket对象是唯一的
var server = net.createServer();
server.listen(PORT, HOST);
console.log('Server listening on ' +
    server.address().address + ':' + server.address().port);

server.on('connection', function(sock) {
    console.log('CONNECTED: ' +
         sock.remoteAddress +':'+ sock.remotePort);
});
```

首先我们来看下 `net.createServer`，它返回一个 `Server` 的实例，如下。

```
1075 function Server(options, connectionListener) {
1076   if (!(this instanceof Server))
1077     return new Server(options, connectionListener);
1078
1079   EventEmitter.call(this);
1080
1081   var self = this;
1082
1083   if (typeof options === 'function') {
1084     connectionListener = options;
1085     options = {};
1086     self.on('connection', connectionListener);
1087   } else {
1088     options = options || {};
1089
1090     if (typeof connectionListener === 'function') {
1091       self.on('connection', connectionListener);
1092     }
1093   }
1094
1095   this._connections = 0;
1096   // ...
1111
1112   this._handle = null;
1113   this._usingSlaves = false;
1114   this._slaves = [];
1115   this._unref = false;
1116
1117   this.allowHalfOpen = options.allowHalfOpen || false;
1118   this.pauseOnConnect = !!options.pauseOnConnect;
1119 }
1120 util.inherits(Server, EventEmitter);
```

`Server` 继承了 `EventEmitter`，如果传入 callback 函数， L1086，L1091 则把传入的函数作为监听者 绑定到 `connnection` 事件上， 然后 listen 。我们看看作为 server 端连接到来的回调处理。

```
1400 function onconnection(err, clientHandle) {
1401   var handle = this;
1402   var self = handle.owner;
1403
1404   debug('onconnection');
1405
1406   if (err) {
1407     self.emit('error', errnoException(err, 'accept'));
1408     return;
1409   }
1410
1411   if (self.maxConnections && self._connections >= self.maxConnections) {
1412     clientHandle.close();
1413     return;
1414   }
1415
1416   var socket = new Socket({
1417     handle: clientHandle,
1418     allowHalfOpen: self.allowHalfOpen,
1419     pauseOnCreate: self.pauseOnConnect
1420   });
1421   socket.readable = socket.writable = true;
1422
1423
1424   self._connections++;
1425   socket.server = self;
1426   socket._server = self;
1427   // ...
1431   self.emit('connection', socket);
1432 }
```

此函数由 `TCPWrap::OnConnection` 回调，`tcp_wrap->MakeCallback(env->onconnection_string(), ARRAY_SIZE(argv), argv);`，第一个参数标识状态，第二个参数为连接的句柄。

L1416-L1421, 根据传过来的句柄，创建 JS 层面的 Socket。并在 L1431 向观察者发送 connection 事件。

上面 TCP 服务端的例子，server 监听connection 事件，自定义用户处理逻辑。

创建TCP客户端

现在让我们创建一个TCP客户端连接到刚创建的服务器上，该客户端向服务器发送一串消息，并在得到服务器的反馈后关闭连接。下面的代码描述了这一过程。

```
var net = require('net');

var HOST = '127.0.0.1';
var PORT = 6969;

var client = new net.Socket();
client.connect(PORT, HOST, function() {

    console.log('CONNECTED TO: ' + HOST + ':' + PORT);
    // 建立连接后立即向服务器发送数据，服务器将收到这些数据
    client.write('I am Chuck Norris!');

});

// 为客户端添加“data”事件处理函数
// data是服务器发回的数据
client.on('data', function(data) {

    console.log('DATA: ' + data);
    // 完全关闭连接
    client.destroy();

});

// 为客户端添加“close”事件处理函数
client.on('close', function() {
    console.log('Connection closed');
});
```

创建 `Socket` 对象后，client 端向server端发起连接，在真正的连接之前，需要进行 DNS 查询（提供 IP 的不用）， 调用 `lookupAndConnect` , 之后才是调用 `function connect(self, address, port, addressType, localAddress, localPort)` 发起连接。

我们注意到五元祖: `<remoteAddress, remotePort, addressType, localAddress, localPort>` , 他们唯一的标识了一个网络连接。

建立起全双工的 Socket 后，用户程序就可以监听 「data」事件，获取数据了。

总结

参考

Crypto

什么是 SSL ？

Secure Sockets Layer，这是其全名，他的作用是协议，定义了用来对网络发出的数据进行加密的格式和规则。

```
        +------+                                                +-----
-+
服务器 | data | -- SSL 加密 --> 发送 --> 接收 -- SSL 解密 -- | data
| 客户端
        +------+                                                +-----
-+

        +------+                                                +-----
-+
服务器 | data | -- SSL 解密 --> 接收 <-- 发送 -- SSL 加密 -- | data
| 客户端
        +------+                                                +-----
-+
```

> 注：TLS 1.0 等同于 SSL 3.1，TLS 1.1 等同于 SSL 3.2，TLS 1.2 等同于 SSL 3.3。

OpenSSL

OpenSSL 是在程序上对 SSL 标准的一个实现，提供了：

- libcrypto 通用加密库
- libssl TLS/SSL 的实现
- openssl 命令行工具

Node.js 是完全采用 OpenSSL 进行加密的，其相关的 TLS HTTPS 服务器模块和 Crypto 加密模块都是通过 C++ 在底层调用 OpenSSL 。

OpenSSL 实现了对称加密:

```
AES(128) | DES(64) | Blowfish(64) | CAST(64) | IDEA(64) | RC2(64
) | RC5(64)
```

非对称加密:

```
DH | RSA | DSA | EC
```

以及一些信息摘要:

```
MD2 | MD5 | MDC2 | SHA | RIPEMD | DSS
```

其中信息摘要是一些采用哈希算法的加密方式，也意味着这种加密是单向的，不能反向解密。这种方式的加密大多是用于保护安全口令，比如登录密码。这里面我们最长用的是 MD5 和 SHA (建议采用更稳定的 SHA1, MD5通过查表大法已经不再单向)。

使用非对称加密会损耗性能，对称加密又不能在网络传输，那该怎么办呢？答案是:结合两者一起使用。 `ssl/tls` 实际上也是如此， `tls` 将两者完美组合使用。

TLS HTTPS 服务器

想要建立一个 Node.js TLS 服务器，需要使用 tls 模块:

```
var Tls = require('tls');
```

在开始搭建服务器之前，我们还有些重要的工作要做，那就是证书，签名证书。

基于 SSL 加密的服务器，在与客户端开始建立连接时，会发送一个签名证书。客户端在自己的内部存储了一些公认的权威证书认证机构，即 CA。客户端通过在自己的 CA 表中查找，来匹配服务器发送的证书上的签名机构，以此来判断面对的服务器是不是一个可信的服务器。

如果这个服务器发送的证书，上面的签名机构不在客户端的 CA 列表中，那么这个服务器很有可能是伪造的，你应该听说过“中间人攻击”。

```
+------------+
| 真正的服务器 |                 选择权：连接？不连接？  +-------+
+------------+               +-----------------  | 客户端 |
                      https  |                   +-------+
+--------------+             | 拦截通信包
| 破坏者的服务器 | ----------+
+--------------+  证书是伪造的
```

总结

加密的安全性，主要是以下两种因素共同决定

使用的加密算法

- 密钥(key)的长度
- 在相同的算法下，密钥长度增加一位，暴力破解的难度指数级增加。如果使用对称加密，现在AES-256足够安全。

不过据说RSA的1024位密钥已经能被政府用非常昂贵的设备暴力破解，所以在使用RSA算法时，密钥长度要选2048，并没有决对的安全。

加密的性能上，

参考

- http://www.jianshu.com/p/a8b87e436ac7

HTTP 1/2

回到我们之前的 「Hello World」例子, 短短数行即可。

```
const http = require('http');
const hostname = '127.0.0.1';
const port = 1337;

http.createServer((req, res) => {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World\n');
}).listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```

因为 Node.js 把许多细节都已在源码中封装好了，主要代码在 lib/*http**.js 这些文件中，现在就让我们照着上述代码，看看从一个 HTTP 请求的到来直到响应，Node.js 都为我们在源码层做了些什么。

Server

在 Node.js 中，若要收到一个 HTTP 请求，首先需要创建一个 http.Server 类的实例，然后监听它的 request 事件。由于 HTTP 协议属于应用层，在下层的传输层通常使用的是 TCP 协议，所以 net.Server 类正是 http.Server 类的父类。

```
// lib/_http_server.js
// ...

function Server(requestListener) {
  if (!(this instanceof Server)) return new Server(requestListen
er);
  net.Server.call(this, { allowHalfOpen: true });

  if (requestListener) {
    this.addListener('request', requestListener);
  }

  // ...
  this.addListener('connection', connectionListener);

  this.addListener('clientError', function(err, conn) {
    conn.destroy(err);
  });

  this.timeout = 2 * 60 * 1000;

  this._pendingResponseData = 0;
}
util.inherits(Server, net.Server);
```

`requestListener` 回调函数作为观察者，监听了 `request` 事件， 默认超时时间为2分钟。

而当连接建立时，观察者 connectionListener 处理 `connection` 事件。

这时，则需要一个 HTTP parser 来解析通过 TCP 传输过来的数据：

```
// lib/_http_server.js
const parsers = common.parsers;
// ...

function connectionListener(socket) {
  // ...
  var parser = parsers.alloc();
  parser.reinitialize(HTTPParser.REQUEST);
  parser.socket = socket;
  socket.parser = parser;
  parser.incoming = null;
  // ...
}
```

HTTP Parser

值得一提的是，parser 是从一个“池”中获取的，这个“池”使用了一种叫做 freelist的数据结构。为了尽可能的对 parser 进行重用，并避免了不断调用构造函数的消耗，且设有数量上限（http 模块中为 1000）。

HTTPParser 的实现目前由 C++绑定实现，具体参见 deps/http_parser 目录。但笔者这边拓展一下：

社区有过对 http_parser 实现性能的争论，性能上 JS 实现的版本超越 C 的实现。

原因是多方面的：

- 去调了 C++ 绑定层。
- JS 实现，避免了 C 栈和 JS 堆栈的切换和参数拷贝。
- V8 JIT 对热点函数的优化。

即便有上述优势，社区目前还是没有合并，处于pending 状态，结合个人和社区观点：

- 并发请求会导致 garbage collection 频繁，触发GC 停顿。
- 可以作为第三方模块存在。

> pull request: https://github.com/nodejs/node/pull/1457/

这里的 parser 也是基于事件的，很符合 Node.js 的核心思想。

```
// lib/_http_common.js
// ...
const binding = process.binding('http_parser');
const HTTPParser = binding.HTTPParser;
const FreeList = require('internal/freelist').FreeList;
// ...

var parsers = new FreeList('parsers', 1000, function() {
  var parser = new HTTPParser(HTTPParser.REQUEST);
  // ...
  parser[kOnHeaders] = parserOnHeaders;
  parser[kOnHeadersComplete] = parserOnHeadersComplete;
  parser[kOnBody] = parserOnBody;
  parser[kOnMessageComplete] = parserOnMessageComplete;
  parser[kOnExecute] = null;

  return parser;
});
exports.parsers = parsers;

// lib/_http_server.js
// ...

function connectionListener(socket) {
  parser.onIncoming = parserOnIncoming;
}
```

所以一个完整的 HTTP 请求从接收到完全解析，会挨个经历 parser 上的如下事件监听器：

- parserOnHeaders：不断解析推入的请求头数据。
- parserOnHeadersComplete：请求头解析完毕，构造 header 对象，为请求体创建 http.IncomingMessage 实例。
- parserOnBody：不断解析推入的请求体数据。
- parserOnExecute：请求体解析完毕，检查解析是否报错，若报错，直接触发 clientError 事件。若请求为 CONNECT 方法，或带有 Upgrade 头，则直接触发 connect 或 upgrade 事件。
- parserOnIncoming：处理具体解析完毕的请求。

前面提到的 `request` 事件到底是在哪里触发的呢？回到源码

```
// lib/_http_server.js
// ...

function connectionListener(socket) {
  var outgoing = [];
  var incoming = [];
  // ...

  function parserOnIncoming(req, shouldKeepAlive) {
    incoming.push(req);
    // ...
    var res = new ServerResponse(req);

    if (socket._httpMessage) {
      outgoing.push(res);
    } else {
      res.assignSocket(socket);
    }

    res.on('finish', resOnFinish);
    function resOnFinish() {
      incoming.shift();
      // ...
      var m = outgoing.shift();
      if (m) {
        m.assignSocket(socket);
      }
    }
    // ...
    self.emit('request', req, res);
  }
}
```

我们注意到 2个队列， `incoming` 和 `outgoing` , 他们用于缓冲 IncomingMessage 实例和对应的 ServerResponse 实例。 通过 `IncomingMessage` 实例构建相应的 `ServerResponse` 实例, 并且通过 `res.assignSocket(socket);` ，绑定了 三元组 `<req, res, socket>` 。

最后，发送 request 事件，参数为req, res。回到 hello world 中，监听者拿到 req 和 res, 向 response 流中写入HTTP头 和内容发送出去。

总结

对象池也是内存池的一种衍生，需要在内存和性能方面折中考量。

上面只是梳理了一个主线，其他异常处理，安全等方面剖析后面的章节会一一解读。

参考

- https://docs.google.com/document/d/1A3cxhZg2aktJeSGt0P-8KrA4WyG1c8LlPomQflaQs8s/edit

HTTP 2/2

http 模块提供了两个函数 http.request 和 http.get，功能是作为客户端向 HTTP服务器发起请求。

这通常来实现自己的爬虫程序， 笔者自己写的一个爬取知乎的一个例子：https://github.com/yjhjstz/iZhihu

GET 例子

```
const http = require("http")
http.get('http://www.baidu.com', (res) => {
  console.log(`Got response: ${res.statusCode}`);
  // consume response body
  res.resume();
}).on('error', (e) => {
  console.log(`Got error: ${e.message}`);
});
```

上面的程序会返回一个200的状态码！

HTTP Client

Node.js 中，http.get 通过创建一个 `ClientRequest` 的对象，建立与服务端的连接通信。

```
// lib/_http_client.js
function ClientRequest(options, cb) {
  var self = this;
  OutgoingMessage.call(self);

  // ...
  const defaultPort = options.defaultPort ||
                      self.agent && self.agent.defaultPort;

  var port = options.port = options.port || defaultPort || 80;
```

```
  var host = options.host = options.hostname || options.host ||
'localhost';

  if (options.setHost === undefined) {
    var setHost = true;
  }

  self.socketPath = options.socketPath;

  var method = self.method = (options.method || 'GET').toUpperCa
se();
  if (!common._checkIsHttpToken(method)) {
    throw new TypeError('Method must be a valid HTTP token');
  }
  self.path = options.path || '/';
  if (cb) {
    self.once('response', cb);
  }

  // ...

  var called = false;
  if (self.socketPath) {
    // ...
  } else if (self.agent) {
    // ...
  } else {
    // No agent, default to Connection:close.
    self._last = true;
    self.shouldKeepAlive = false;
    if (typeof options.createConnection === 'function') {
      const newSocket = options.createConnection(options, oncrea
te);
      if (newSocket && !called) {
        called = true;
        self.onSocket(newSocket);
      } else {
        return;
      }
    } else {
```

```
        debug('CLIENT use net.createConnection', options);
        self.onSocket(net.createConnection(options));
      }
    }

    function oncreate(err, socket) {
      // ...
    }

    self._deferToConnect(null, null, function() {
      self._flush();
      self = null;
    });
  }

  util.inherits(ClientRequest, OutgoingMessage);
```

callback 通过 `self.once('response', cb);`，监听了 response 事件。之后如果没有设置代理服务，则默认使用 net 模块创建与服务器的连接。那么 response 事件是哪里发送的呢？

下面我们看到比较重要的 `onSocket` 函数。

```
ClientRequest.prototype.onSocket = function(socket) {
  process.nextTick(onSocketNT, this, socket);
};

function onSocketNT(req, socket) {
  if (req.aborted) {
    // If we were aborted while waiting for a socket, skip the w
hole thing.
    socket.emit('free');
  } else {
    tickOnSocket(req, socket);
  }
}
```

这边 onSocket 必须是一个异步函数，大家可以仔细体会下！ 同时 onSocketNT 会异常做了处理，当请求失败时，则发送 free 事件。 否则来到 `tickOnSocket` 。

```
function tickOnSocket(req, socket) {
  var parser = parsers.alloc();
  req.socket = socket;
  req.connection = socket;
  parser.reinitialize(HTTPParser.RESPONSE);
  parser.socket = socket;
  parser.incoming = null;
  parser.outgoing = req;
  req.parser = parser;

  socket.parser = parser;
  socket._httpMessage = req;

  // Setup "drain" propagation.
  httpSocketSetup(socket);

  // Propagate headers limit from request object to parser
  if (typeof req.maxHeadersCount === 'number') {
    parser.maxHeaderPairs = req.maxHeadersCount << 1;
  } else {
    // Set default value because parser may be reused from FreeL
ist
    parser.maxHeaderPairs = 2000;
  }

  parser.onIncoming = parserOnIncomingClient;
  socket.removeListener('error', freeSocketErrorListener);
  socket.on('error', socketErrorListener);
  socket.on('data', socketOnData);
  socket.on('end', socketOnEnd);
  socket.on('close', socketCloseListener);
  req.emit('socket', socket);
}
```

同 HTTP Server 类似，从池中申请一个解析器，用于解析 HTTP 协议， 到这一步说明连接已经建立，所以重新设置 error 事件的回调。

同时设置 数据回调等，然后发送 `socket` 事件, 来到 `parserOnIncomingClient` .

```
// client
function parserOnIncomingClient(res, shouldKeepAlive) {
  var socket = this.socket;
  var req = socket._httpMessage;


  // propagate "domain" setting...
  if (req.domain && !res.domain) {
    debug('setting "res.domain"');
    res.domain = req.domain;
  }

  debug('AGENT incoming response!');

  if (req.res) {
    // We already have a response object, this means the server
    // sent a double response.
    socket.destroy();
    return;
  }
  req.res = res;

  var isHeadResponse = req.method === 'HEAD';

  // ...
  req.res = res;
  res.req = req;

  // add our listener first, so that we guarantee socket cleanup
  res.on('end', responseOnEnd);
  var handled = req.emit('response', res);

  // If the user did not listen for the 'response' event, then t
hey
  // can't possibly read the data, so we ._dump() it into the vo
id
  // so that the socket doesn't hang there in a paused state.
  if (!handled)
    res._dump();
```

```
  return isHeadResponse;
}
```

在这里发送 response 事件，参数对象 `res` 上也挂上了 `req` 对象。这样 req 和 res 就相互引用。

用户的 callback 终于得到回调。

总结

上面只是梳理了一个http client 主线， 实际我们很少使用该模块，而是使用第三方的 npm 包，比如

- urllib (轻量级)
- request

参考

Chapter11

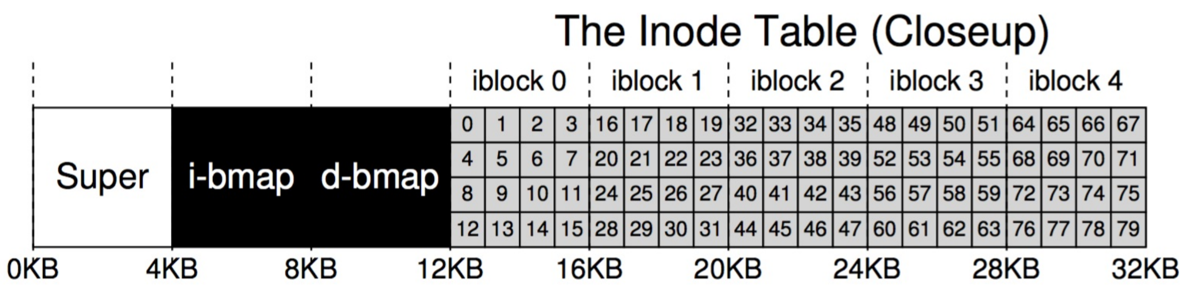
文件系统实现原理

文件系统存储在磁盘上，一般磁盘会被划分为多个分区，每个分区可以是一个独立的文件系统。

磁盘的0号扇区为主引导记录(Master Boot Record，MBR)，用来引导计算机。

在MBR的结尾是分区表，该分区表标识了每个分区的起始和结束地址。表中的一个分区被标识为活动分区，在计算机被引导时，BIOS读入并执行MBR。MBR做首先做的是确定活动分区，读入它的第一个块，称为引导块，并执行。

一种常见的文件系统（分区）结构图



- Super块
  - 在系统启动时，会读取Super块，它包含了文件系统的所要重要参数, 通常会做多个备份。
- i-bmap 块
  - 用来管理inode，标识inode是否空闲或被使用了。
- d-bmap块
  - 用来管理磁盘块，标识磁盘块是空闲还是被使用了。

文件系统实现

文件系统一个重要的功能是记录文件使用了哪些磁盘块以及磁盘块的管理，标识磁盘块是否被使用，还是空闲。实现思路有下面几种。

- 连续分配

  - 把每个文件存储在相邻的磁盘块上。如磁盘块大小2KB,200KB的文件，需要 100个磁盘块，如果磁盘块大小为4KB,刚需要50个磁盘块。

由于每个文件都是从一个新的磁盘块开始的，这样如果一个文件只占了磁盘块大小的一半，那么另一半就被浪费了，没法被别的文件使用，不过连续分配的实现，实现比较简单，只需要记录文件的第一个磁盘块位置和块数，另外读取性能，因为只需要一次寻道，之后不需要导道和旋转延迟。

随着时间推移，磁盘碎片比较严重。因为反复写文件，删除文后，容易在磁盘块上形成空洞。

- 链表分配
  - 为每个文件构造磁盘块链表，每个块在前面指向文件的下一个磁盘块。

    因为是一个链表，所以顺序读取很快，但随机读取很慢，且每个块中，指向下一个磁盘块的指针是要占用空间的，这样导致每个磁盘块能够存储的数据不再是2的整数次幂。

    但程序读写文件一般是以2的整数次幂来读写磁盘，这样造成额外的开销，因为读一个块的数据，要读取二个磁盘块。
- inode 节点方案
  - 使用一个特殊的东西来记录每个文件的所使用的磁盘块， 这特殊的东西称为i节点数据结构，其存储了文件 一些属性及文件所使用的到的磁盘块。

    i节点只有在对应的文件被打开时，才会存在内存中，这样即使文件系统文件非常多，只要打开的文件不多，就不会占用太多的内存。

    文件的元数据和文件数据是分开存储，也就是一个文件有i节点和数据文件这两个属性。

    每个存储i节点的磁盘块空间是有限，且一个文件只有一个i节点，那么当一个文件比较大时，怎么解决？大家想没有想起C语言中的指针及指针的指针。

    一种解决办法是预留部分数据块，用来存储指向磁盘块的指针，而不是直接直接指向磁盘块。

Linux文件系统的实现

Unix/Linux文件系统的实现是采用i节点的方案，文件系统ext2、ext3都是如此，ext4相比前面二种文件 ，做了不少优化，本书便不在此展开。

文件系统与操作系统的协作

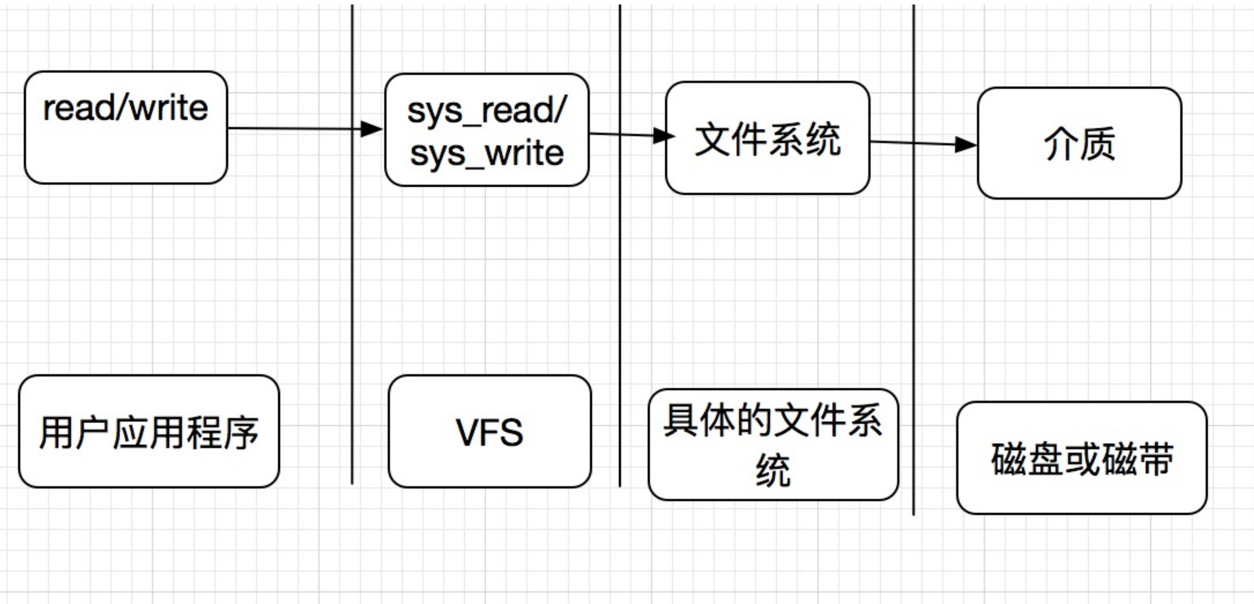
文件系统是操作系统的一部分，操作系统是相对稳定，但文件系统却是有好多，如ext2、ext3 、ext4 、ZFS 等，那么操作系统是如何兼容这些不同的文件系统，以对外提供统一服务。

C++、JAVA程序员很容易想到利用多态来实现，对，操作系统也是采用类似的思路。对不同的文件系统，抽象出所有文件系统都支持的、基本的、概念上的数据结构和接口 ，如前文描述的关于文件和目录的基本操作。

这些统一抽象组件，叫着虚拟文件系统(Virtual File System ,VFS),VFS作为系统内核组件，为用户空间程序提供了文件和文件系统相关的接口。

VFS使用得用户可以直接调用write、read 这样的文件系统调用，而不用考虑底层的文件是什么文件系统。

横向地看下它们的关系，如下图所示



总结

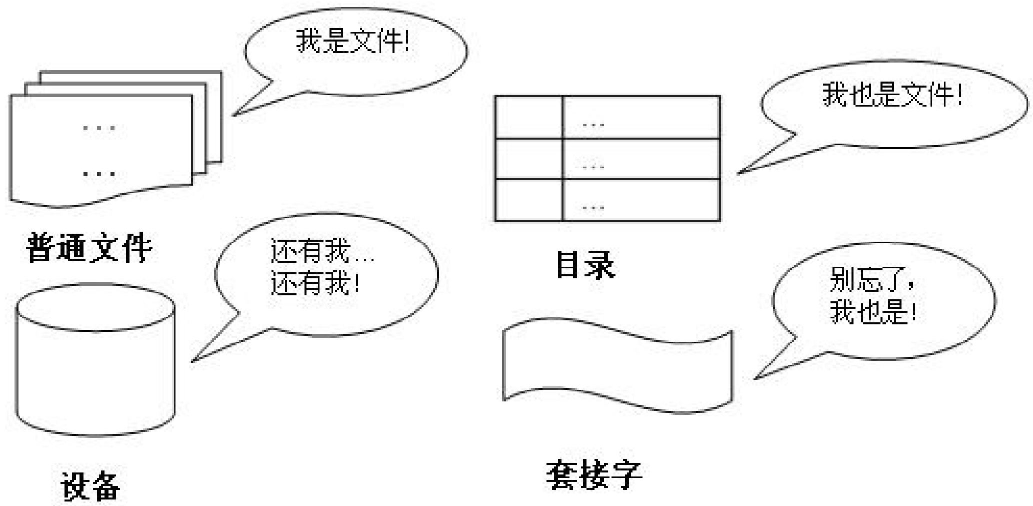
文件系统是对磁盘设备的抽象，屏蔽了具体的存储类型，比如磁盘，内存。

文件抽象

fs模块是文件操作的封装，它提供了文件的读取、写入、更名、删除、遍历目录、链接POSIX文件系统操作。与其他模块不同的是，fs模块中的所有操作都提供了异步和同步两个版本，例如读取文件内容函数的异步方法：readFile(),同步方法readFileSync()。

一切皆文件

“一切皆是文件”是 Unix/Linux 的基本哲学之一。不仅普通的文件，目录、字符设备、块设备、 套接字等在 Unix/Linux 中都是以文件被对待；它们虽然类型不同，但是对其提供的却是同一套操作接口。



文件是一种抽象机制，它对磁盘进行了抽象。

文件就是字节序列，每个I/O设备，包括磁盘、键盘、显示器、甚至网络，都可以抽象成文件，在Unix/Linux系统中，系统中所有的输入输出都是通过调用IO系统调用来完成。

文件是对IO的抽象，就像虚拟存储器是对程序存储的抽象，进程是对一个正在运行程序的抽象。这些都是操作系统重要的抽象。

抽象机制最重要的特性是对管理对象的命名，所以文件有文件名，且文件名要符合一定的规范。

文件主要操作

- open
- read
- write
- close 上面的操作比较简单，就不是细说，后面会写文章再介绍读文件、写文件、刷新数据这几个重要的操作。如果有兴趣，可以通过man 2 read 命令来查看帮助文档。

文件类型

可以通过ls -l查看文件类型，主要有下面几种常见的。

- 普通文件
  - 包括文本文件和二进制文件
- 目录
  - 和普通文件相比，目录也存储在介质上，但是目录不存储常规文件，它只是用来组织、管理文件。
- proc文件
  - proc不存储，所以不占用任何空间，proc使得内核可以生成与系统状态和配置相关的信息，该信息可以由用户和系统内核从普通文件读取，无需专门的工具。 其它更多的文件类型，可以通过man ls 查看。

文件属性

文件属性包括文件权限信息、创建时间、最后修改时间、最后读取时间、文件大小、文件引用数等信息，这些文件属性也称为文件元数据。

文件系统之高级读写

文件映射 mmap

`man 2 mmap` 查看：

```
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flags, int fd,
off_t offset);
```

通过mmap系统调用，把一个文件映射到进程虚拟址址空间上。也就是说磁盘上的文件，现在在在系统看来是一个内存数组了，这样应用程序访问文件就不需要系统IO调用，而是直接读取内存。

优点：

- 1、从内存映像文件中读写，避免了read、write多余的拷贝。
- 2、从内存映像文件中读写，避免了多余的系统调用和用户-内核模式的切换
- 3、可以多个进程共享内存映像文件。

缺点：

- 1、内存映像需要整数倍页大小，如果文件较小，会浪费内存。
- 2、内存映像需要进程地址空间，大的内存映像可能导致地址空间碎片，找不到足够大的空余连续区域供其它用。

离散 I/O

readv和writev函数让我们在单个函数调用里从多个不连续的缓冲里读入或写出。这些操作被称为分散读（scatter read）和集合写（gather write）。

```
#include <sys/uio.h>

ssize_t readv(int filedes, const struct iovec *iov, int iovcnt);

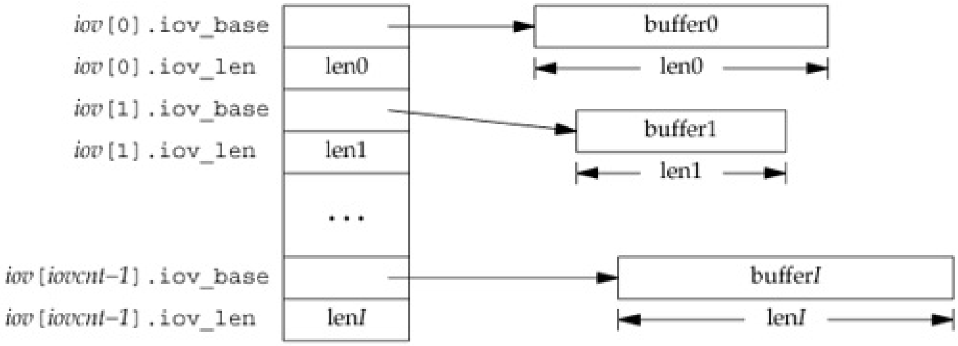
ssize_t writev(int filedes, const struct iovec *iov, int iovcnt)
;

两者都返回读或写的字节数，错误返回-1。
```

这两个函数的第二个参数是指向iovec结构数组的一个指针：

```
struct iovec {
  void    *iov_base;      /* starting address of buffer */
  size_t iov_len;         /* size of buffer */
};
```

iov数组中的元素数由iovcnt说明。其最大值受限于IOV_MAX。



writev以顺序iov[0]，iov[1]至iov[iovcnt-1]从缓冲区中聚集输出数据。writev返回输出的字节总数，通常，它应等于所有缓冲区长度之和。

readv则将读入的数据按上述同样顺序散布到缓冲区中。readv总是先填满一个缓冲区，然后再填写下一个。readv返回读到的总字节数。如果遇到文件结尾，已无数据可读，则返回0。

总结

- 零拷贝技术可以减少数据拷贝和共享总线操作的次数，消除传输数据在存储器之间不必要的中间拷贝次数，从而有效地提高数据传输效率。而且，零拷贝技术减少了用户应用程序地址空间和操作系统内核地址空间之间因为上下文切换而带来的开销。

是用户程序尝试优化的重要可选的优化手段。

- 向量 I/O 操作可以取代多个线性 I/O 操作, 性能更好
  - 除了减少了发起的系统调用次数，通过内部优化，向量 I/O 可以比线性 I/O 提供更好的性能。
  - 支持原子性, 一个进程可以执行单个向量 I/O 操作，避免了和其他进程交叉操作的风险。

参考

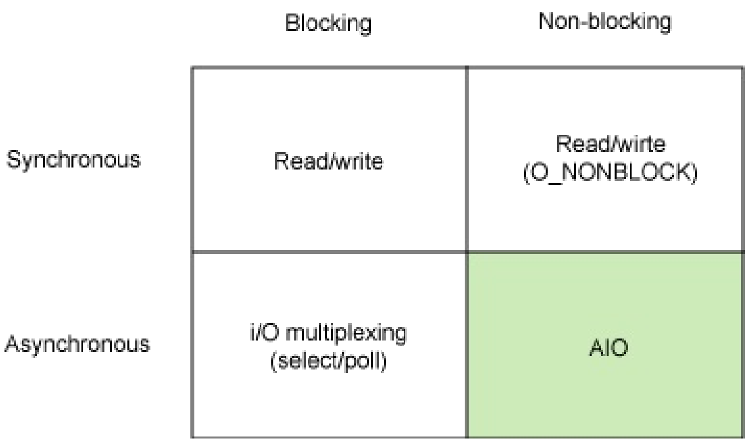
http://www.ibm.com/developerworks/cn/linux/l-cn-zerocopy1/

异步那些事儿

Linux 异步 I/O 是 Linux 内核中提供的一个相当新的增强。它是 2.6 版本内核的一个标准特性，AIO 背后的基本思想是允许进程发起很多 I/O 操作，而不用阻塞或等待任何操作完成。稍后或在接收到 I/O 操作完成的通知时，进程就可以检索 I/O 操作的结果。

I/O 模型

在深入介绍 AIO API 之前，让我们先来探索一下 Linux 上可以使用的不同 I/O 模型。这并不是一个详尽的介绍，但是我们将试图介绍最常用的一些模型来解释它们与异步 I/O 之间的区别。图 1 给出了同步和异步模型，以及阻塞和非阻塞的模型。

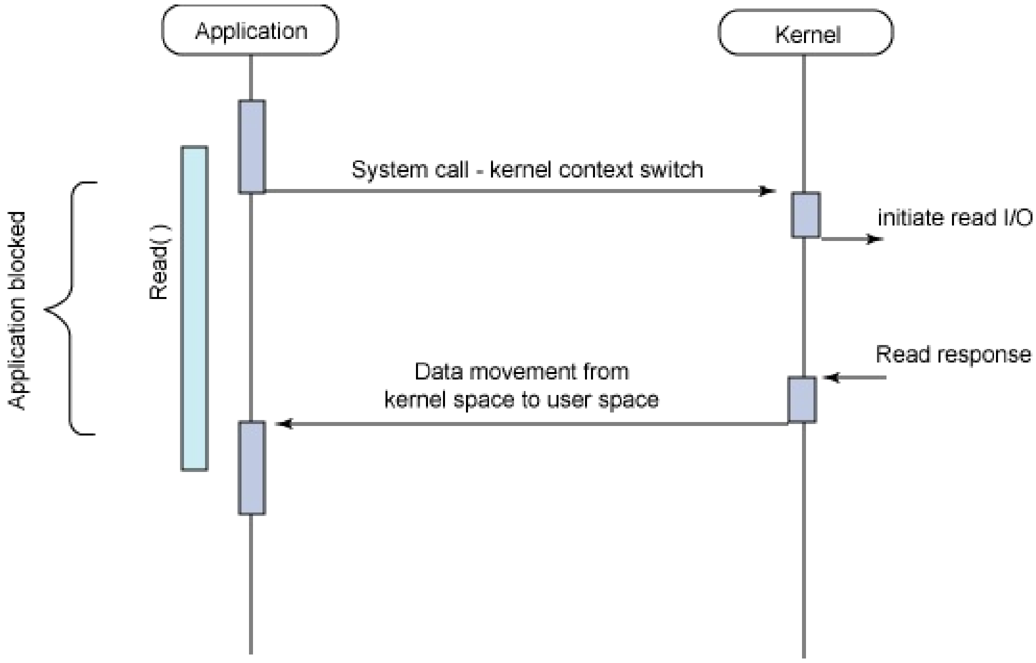


每个 I/O 模型都有自己的使用模式，它们对于特定的应用程序都有自己的优点。本节将简要对其一一进行介绍。

同步阻塞 I/O

最常用的一个模型是同步阻塞 I/O 模型。在这个模型中，用户空间的应用程序执行一个系统调用，这会导致应用程序阻塞。这意味着应用程序会一直阻塞，直到系统调用完成为止（数据传输完成或发生错误）。调用应用程序处于一种不再消费 CPU 而只是简单等待响应的状态，因此从处理的角度来看，这是非常有效的。 图 2 给出了传统的阻塞 I/O 模型，这也是目前应用程序中最为常用的一种模型。在调用 read 系统调用时，应用程序会阻塞并对内核进行上下文切换。然后会触发读操作，当响

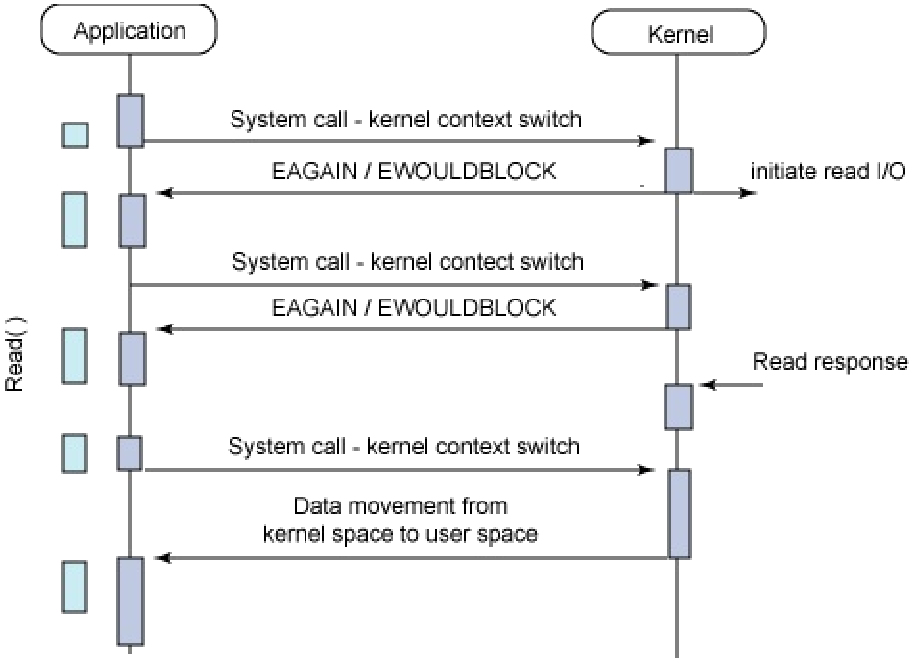
应返回时（从我们正在从中读取的设备中返回），数据就被拷贝到用户空间的缓冲区中。然后应用程序就会解除阻塞（read 调用返回）。



从应用程序的角度来说，read 调用会延续很长时间。实际上，在内核执行读操作和其他工作时，应用程序的确会被阻塞。

同步非阻塞 I/O

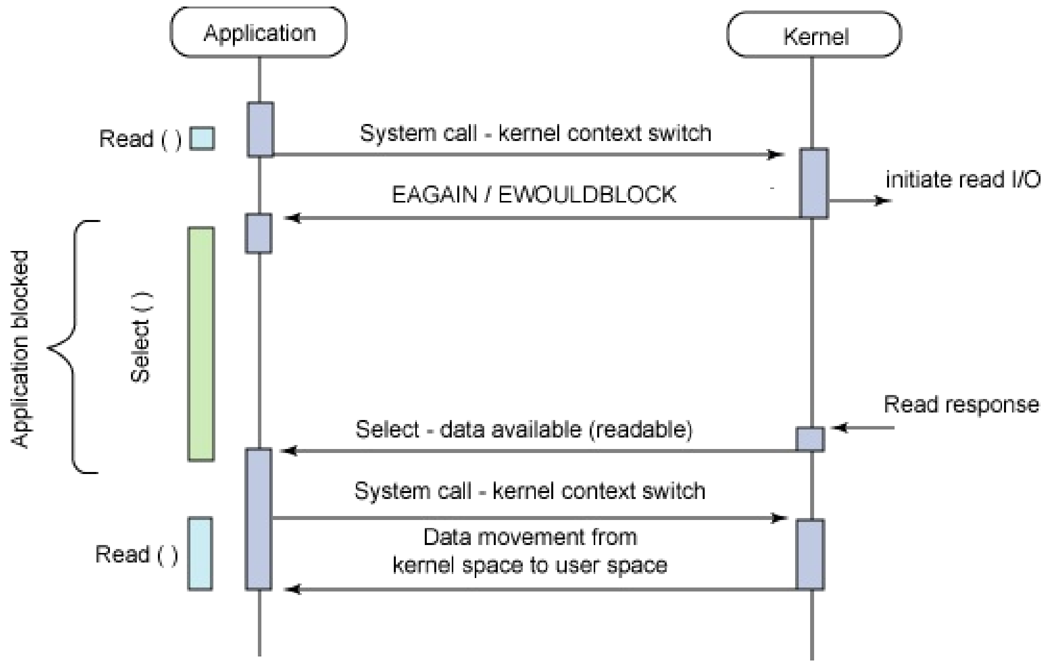
同步阻塞 I/O 的一种效率稍低的变种是同步非阻塞 I/O。在这种模型中，设备是以非阻塞的形式打开的。这意味着 I/O 操作不会立即完成，read 操作可能会返回一个错误代码，说明这个命令不能立即满足（EAGAIN 或 EWOULDBLOCK），如图 3 所示。



非阻塞的实现是 I/O 命令可能并不会立即满足，需要应用程序调用许多次来等待操作完成。这可能效率不高，因为在很多情况下，当内核执行这个命令时，应用程序必须要进行忙碌等待，直到数据可用为止，或者试图执行其他工作。正如图 3 所示的一样，这个方法可以引入 I/O 操作的延时，因为数据在内核中变为可用到用户调用 read 返回数据之间存在一定的间隔，这会导致整体数据吞吐量的降低。

异步阻塞 I/O

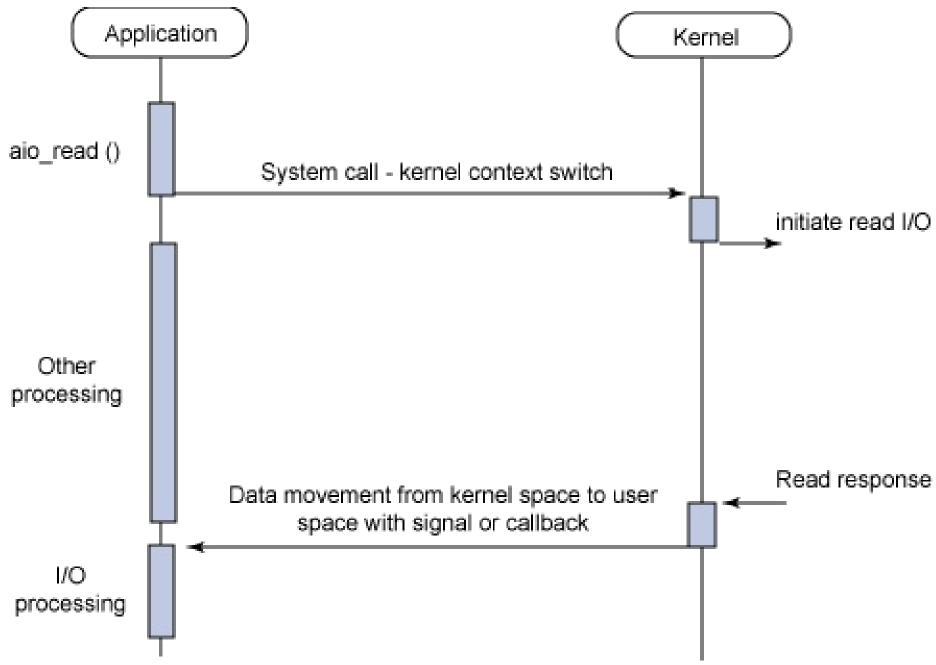
另外一个阻塞解决方案是带有阻塞通知的非阻塞 I/O。在这种模型中，配置的是非阻塞 I/O，然后使用阻塞 select 系统调用来确定一个 I/O 描述符何时有操作。使 select 调用非常有趣的是它可以用来为多个描述符提供通知，而不仅仅为一个描述符提供通知。对于每个提示符来说，我们可以请求这个描述符可以写数据、有读数据可用以及是否发生错误的通知。



select 函数所提供的功能（异步阻塞 I/O）与 AIO 类似。不过，它是对通知事件进行阻塞，而不是对 I/O 调用进行阻塞。

异步非阻塞 I/O

异步非阻塞 I/O 模型是一种处理与 I/O 重叠进行的模型。读请求会立即返回，说明 read 请求已经成功发起了。在后台完成读操作时，应用程序然后会执行其他处理操作。当 read 的响应到达时，就会产生一个信号或执行一个基于线程的回调函数来完成这次 I/O 处理过程。



在一个进程中为了执行多个 I/O 请求而对计算操作和 I/O 处理进行重叠处理的能力利用了处理速度与 I/O 速度之间的差异。当一个或多个 I/O 请求挂起时，CPU 可以执行其他任务；或者更为常见的是，在发起其他 I/O 的同时对已经完成的 I/O 进行操作。

总结

慢速的 IO 设备和高速的 CPU 如何协作是门学问。 但这里并不是说同步阻塞IO 一定不好，还是要根据场景灵活选择。

依照作者的经验， 同步 IO 适用于时间可控，少量调用的场景。 异步 IO 适用于时间不可控（如网络异常），大量调用的场景。

参考

- https://www.ibm.com/developerworks/cn/linux/l-async/

libuv 选型

linux native aio

Linux native aio 有两种API，一种是libaio提供的API，一种是利用系统调用封装成的API,后者使用的较多，因为不需要额外的库且简单。

- io_setup : 是用来设置一个异步请求的上下文，第一个参数是请求事件的个数，第二个参数唯一标识一个异步请求。
- io_commit: 是用来提交一个异步io请求的，在提交之前，需要设置一下结构体 `iocb` 。
- io_getevents: 用来获取完成的io事件，参数 `min_nr` 是事件个数的的最小值， `nr` 是事件个数的最大值，如果没有足够的事件发生，该函数会阻塞。
- io_destroy：在所有时间处理完之后，调用此函数销毁异步io请求。

限制

aio只能使用于常规的文件IO，不能使用于socket，管道等IO，但对于 libuv 的 fs 模块使用需求已经足够了。

io_getevents在调用之后会阻塞直到有足够的事件发生，因此要实现真正的异步IO，需要借助eventfd和epoll达到目的。

libuv native aio 实现

笔者实现过一个基于 libuv 的 native aio，https://github.com/yjhjstz/libuv/commit/2748728635c4f74d6f27524fd36e680a88e4f04a

从理论上看，在libuv中实现AIO,

- 其一：比原来的libuv实现少了一次write系统调用，无需在用户态实现线程池和工作队列.
- 其二：native aio实现可以实现批量回调。

我们看下性能对比数据, 测试脚本是简单的文件读取：

- Threadpool 模型

```
jiangling@young:~/workspace/libuv$ wrk -t4 -c100 -d30s http:
//127.0.0.1:30003/
Running 30s test @ http://127.0.0.1:30003/
4 threads and 100 connections
Thread Stats Avg Stdev Max +/- Stdev
Latency 16.77ms 1.14ms 31.68ms 86.68%
Req/Sec 1.51k 162.66 2.08k 81.34%
178925 requests in 30.00s, 104.26MB read
Requests/sec: 5963.45
Transfer/sec: 3.47MB
```

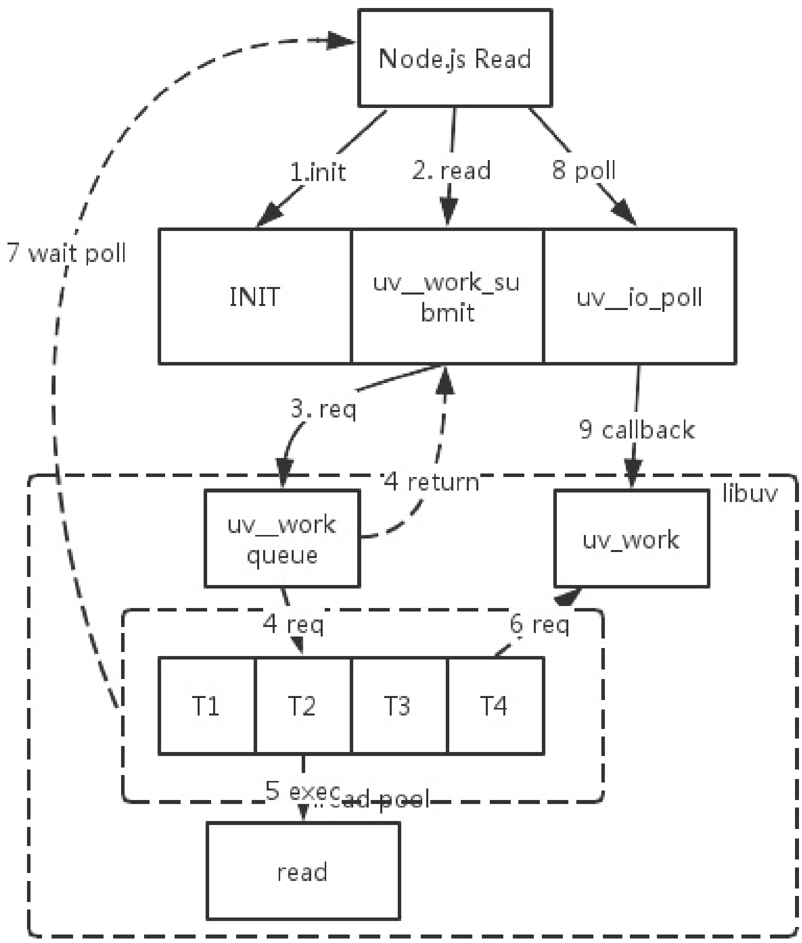
- Native AIO 模型

```
jiangling@young:~/workspace/libuv$ wrk -t4 -c100 -d30s http:
//127.0.0.1:30003/
Running 30s test @ http://127.0.0.1:30003/
4 threads and 100 connections
Thread Stats Avg Stdev Max +/- Stdev
Latency 16.22ms 0.95ms 26.39ms 88.12%
Req/Sec 1.57k 191.14 2.08k 68.50%
185084 requests in 30.00s, 107.85MB read
Requests/sec: 6169.28
Transfer/sec: 3.59MB
```

Max Latency减小**16%**，**tps**提升**3%**。

Threadpool 模型

我们先看下一次 node.js read 的调用示意图：



代码的运行经历了以下步骤：

- 1 node, libuv 初始化；
- 2 node_file.cc中的Read方法调用libuv（fs.c）的 `uv_fs_read` ，封装请求；
- 3 libuv 将请求封装成 uv_work, 提交到任务队列尾部，触发信号；
- 4 此时主线程的read调用返回。
- 5 线程池从uv_work队列中取出一个请求，开始执行read IO；
- 6 向主线程发送信号表明任务完成，等待执行read调用后的其它操作。
- 7 主线程 epoll，从响应队列取已经完成的请求;
- 8 主线程响应 epoll事件；

- 9 主线程执行请求的callback函数。

node.js 异步 IO 的脉络已经清晰，我们清楚的看到这样的一个 Threadpool 模型是全平台适用的。

> Linux 上的 AIO AIO 在 2.5 版本的内核中首次出现，现在已经是 2.6 版本的产品内核的一个标准特性了。

并且由于 Native AIO 是在 linux 2.6之后引入，并且并不稳定。 社区也有过激烈的讨论：

- https://github.com/libuv/libuv/issues/28
- https://github.com/libuv/libuv/issues/461

权衡再三，笔者也非常支持社区采用的模型，赋予用户更多的选择性和可靠性。

总结

用户态的线程池实现给了用户更大的灵活性和选择性。比如：

- 1.线程池的个数，默认是4个，用户可以通过设置环境变量 `UV_THREADPOOL_SIZE` 指定。
- 2.和耗时的 GETADDRINFO 复用线程池。

需要指出的是，线程池模型还有改进的空间：

- `static uv_mutex_t mutex;` 全局锁的优化；
- 支持任务优先级。

参考

文件io

上一章节在讲述了线程池的模型，读者对一次异步 IO 发起到结束有了一个大致的认识。

fs 模块还提供了同步接口，如 `readFileSync` ，这在异步模型的 node.js 的 核心模块中是极为少见的。

请求对象

- 异步读文件接口定义： `fs.readFile = function(path, options, callback_)`
- 同步读文件接口定义： `fs.readFileSync = function(path, options)`

两者明显的差异在于第三个参数 `callback_` , 异步会提交请求然后等待回调，同步则阻塞直到返回。

让我们来看看第三个参数对实现的影响。

```
// fs.js
// ...
var context = new ReadFileContext(callback, encoding);
var req = new FSReqWrap();
req.context = context;
req.oncomplete = readFileAfterOpen;
```

异步的实现中会创建一个请求对象，并且绑定回调和上下文环境。 该请求对象由 C++ 绑定导出。

FSReqWrap (node.js)

FSReqWrap 是由 `src/node_file.cc` 实现并导出，提供给 javascript 使用。

```
class FSReqWrap: public ReqWrap<uv_fs_t> {
 public:
  enum Ownership { COPY, MOVE };

  inline static FSReqWrap* New(Environment* env,
                                Local<Object> req,
                                const char* syscall,
                                const char* data = nullptr,
                                Ownership ownership = COPY);

  inline void Dispose();

  //...
};
```

FSReqWrap 继承 ReqWrap, ReqWrap 是个模板类， `T req_;` 存储了不同类型的请求，在这里 模板编译后， `uv_fs_t req_;` , req_ 存储了 uv_fs_t 请求对象。 这源自于一次性能优化的提交。

> fs: improve `readFile` performance
>
> This commit improves `readFile` performance by reducing number of closure allocations and using `FSReqWrap` directly.

具体了解, https://github.com/iojs/io.js/pull/718 。

在js 层发起请求后，会来到C++绑定层，

```
#define ASYNC_DEST_CALL(func, req, dest, ...)
              \
  Environment* env = Environment::GetCurrent(args);
              \
  CHECK(req->IsObject());
              \
  FSReqWrap* req_wrap = FSReqWrap::New(env, req.As<Object>(), #f
unc, dest);   \
  int err = uv_fs_ ## func(env->event_loop(),
              \
                          &req_wrap->req_,
              \
```

```
                            __VA_ARGS__,
             \
                            After);
             \
  req_wrap->Dispatched();
             \
  if (err < 0) {
             \
    uv_fs_t* uv_req = &req_wrap->req_;
             \
    uv_req->result = err;
             \
    uv_req->path = nullptr;
             \
    After(uv_req);
             \
    req_wrap = nullptr;
             \
  } else {
             \
    args.GetReturnValue().Set(req_wrap->persistent());
             \
  }

#define ASYNC_CALL(func, req, ...)
             \
  ASYNC_DEST_CALL(func, req, nullptr, __VA_ARGS__)
             \
```

这里才会生成 libuv 所需的请求对象， 对于读请求调用 `uv_fs_read` , 提交请求， 指定回调函数为 `After` 。

uv_fs_t (libuv)

看一下libuv的异步读文件代码，deps/uv/src/unix/fs.c：

```
/* uv_fs_t is a subclass of uv_req_t. */
struct uv_fs_s {
  UV_REQ_FIELDS
  uv_fs_type fs_type;
  uv_loop_t* loop;
  uv_fs_cb cb;
  ssize_t result;
  void* ptr;
  const char* path;
  uv_stat_t statbuf;  /* Stores the result of uv_fs_stat() and u
v_fs_fstat(). */
  UV_FS_PRIVATE_FIELDS
};
```

```
#define INIT(subtype)
              \
  do {
              \
    req->type = UV_FS;
              \
    if (cb != NULL)
              \
      uv__req_init(loop, req, UV_FS);
              \
    req->fs_type = UV_FS_ ## subtype;
              \
    req->result = 0;
              \
    req->ptr = NULL;
              \
    req->loop = loop;
              \
    req->path = NULL;
              \
    req->new_path = NULL;
              \
    req->cb = cb;
              \
  }
              \
  while (0)
```

可以看到一次异步文件读操作在libuv层被封装到一个uv_fs_t的结构体，req->cb是来自上层的回调函数（node C++层：src/node_file.cc 的After函数）。

异步io请求最后调用uv__work_submit，把异步io请求提交给线程池。这里有两个函数：

- `uv__fs_work` ：这个是文件io的处理函数，可以看到当cb为NULL的时候，即非异步模式，`uv__fs_work` 在当前线程（事件循环所在线程）直接被调用。如果cb != NULL，即文件io为异步模式，此时把 `uv__fs_work` 和 `uv__fs_done` 提交给线程池。

- `uv__fs_done` ：这个是异步文件io结束后的回调函数。在uv__fs_done里面会回调上层C++模块的cb函数（即req->cb）。

需要特别注意的是：此时**io**操作的主体 **`uv__fs_work`** 函数是在线程池里执行的。但是 **`uv__fs_done`** 必须在事件循环的线程里被回调，因为这个函数最终会回调到用户**js**代码的回调函数，而**js**代码里的所有代码必须在同个线程里面。

线程池的请求对象 —— struct uv__work

先看下 `uv__work` 的定义：

```
struct uv__work {
  void (*work)(struct uv__work *w);
  void (*done)(struct uv__work *w, int status);
  struct uv_loop_s* loop;
  void* wq[2];
};
```

再看看 `uv__work_submit` 做了什么：

```
static void post(QUEUE* q) {
  uv_mutex_lock(&mutex);
  QUEUE_INSERT_TAIL(&wq, q);
  if (idle_threads > 0)
    uv_cond_signal(&cond);
  uv_mutex_unlock(&mutex);
}

void uv__work_submit(uv_loop_t* loop,
                     struct uv__work* w,
                     void (*work)(struct uv__work* w),
                     void (*done)(struct uv__work* w, int status
)) {
  uv_once(&once, init_once);
  w->loop = loop;
  w->work = work;
  w->done = done;
  post(&w->wq);
}
```

`uv__work_submit` 把传进来的 `uv__fs_work` 、 `uv__fs_done` 封装到 `uv__work` 结构体里面，这个结构体表示一个线程操作的请求。通过post把请求提交给线程池。

看到post函数里面的QUEUE_INSERT_TAIL，把该 `uv__work` 对象加进 `wq` 链表里面。wq是一个全局静态变量。也就是说，进程空间里的所有线程共用同一个wq链表。

看到post函数通过uv_cond_signal向相应的条件变量——cond发送信号，处在uv_cond_wait挂起等待的工作线程当中的某个线程被激活。

worker线程往下执行，从wq取出w，执行w->work()。

工作线程完成任务后，调用uv_async_send通知主线程统一的io观察者，执行callback。

回调

总结

通过对各层请求对象的梳理，也详细梳理出了一次 read 请求的脉络, 使读者有了一个理性的认识。

参考

FAQ

同步 require()

```
// Native extension for .js
Module._extensions['.js'] = function(module, filename) {
  var content = fs.readFileSync(filename, 'utf8');
  module._compile(internalModule.stripBOM(content), filename);
};
```

大家可能都有疑问：为什么会选择使用同步而不用异步实现呢？

之所以同步是 Node.js 所遵循的 CommonJS 的模块规范要求的, 具体来说

在当年，CommonJS 社区对此就有很多争议，导致了坚持异步的 AMD 从 CommonJS 中分裂出来。

CommonJS 模块是同步加载和同步执行，AMD 模块是异步加载和异步执行，CMD（Sea.js）模块是异步加载和同步执行。ES6 的模块体系最后选择的是异步加载和同步执行。也就是 Sea.js 的行为是最接近 ES6 模块的。不过 Sea.js 这样做是需要付出代价的——需要扫描代码提取依赖，所以它不像 CommonJS/AMD 是纯运行时的模块系统。

注意 Sea.js 是 2010年之后开发的，提出 CMD 更晚。Node.js 当年（2009年）只有 CommonJS 和 AMD 两个选择。就算当时已经有 CMD 的等价提案，从性能角度出发，Node.js 不太可能选择需要静态分析开销的 类 CMD 方案。考虑到 Node.js 的模块是来自于本地文件系统，最后 Node.js 选择了看上去更简单的 CommonJS 模块规范，直到今天。

- 从模块规范的角度来看，依赖的同步获取是几乎所有模块机制的首选，是符合由无数的语言奠定的开发者的直觉。
- 从模块本身的特性来说的，结论就是使用异步的require收益很小，同时对开发者并不友好。

fs.realpath 缓存

如今的 `realpath` 的实现变得非常简洁，直接调用系统调用realpath。

```
fs.realpath = function realpath(path, options, callback) {
  if (!options) {
    options = {};
  } else if (typeof options === 'function') {
    callback = options;
    options = {};
  } else if (typeof options === 'string') {
    options = {encoding: options};
  } else if (typeof options !== 'object') {
    throw new TypeError('"options" must be a string or an object'
);
  }
  callback = makeCallback(callback);
  if (!nullCheck(path, callback))
    return;
  var req = new FSReqWrap();
  req.oncomplete = callback;
  binding.realpath(pathModule._makeLong(path), options.encoding,
 req);
  return;
};
```

大家可能又有疑问了， 原本提升性能的路径缓存去哪里了，不是说缓存都是提升性能的重要手段吗？

社区的修改可以在 https://github.com/nodejs/node/pull/3594 看到，

> fs: optimize realpath using uv_fs_realpath()
>
> Remove realpath() and realpathSync() cache. Use the native uv_fs_realpath() which is faster then the JS implementation by a few orders of magnitude

去掉了缓存反而提升了性能， 作者的 commit 提交也写的非常清楚：native uv_fs_realpath 实现要大大优于js层的实现， 但并没有说具体原因。

前面我已经提到过了文件系统的基本原理和大致实现，VFS中引入了高速磁盘缓存的机制，这属于一种软件机制，允许内核将原本存在磁盘上的某些信息保存在RAM中，以便对这些数据的进一步访问能快速进行，而不必慢速访问磁盘本身。 高速磁盘缓存可大致分为以下三种：

- 目录项高速缓存——主要存放的是描述文件系统路径名的目录项对象
- 索引节点高速缓存——主要存放的是描述磁盘索引节点的索引节点对象
- 页高速缓存——主要存放的是完整的数据页对象，每个页所包含的数据一定属于某个文件，同时，所有的文件读写操作都依赖于页高速缓存。其是Linux内核所使用的主要磁盘高速缓存。

`readpath` 的 native 实现的高性能得益于目录项高速缓存，有自身的淘汰机制，保持自身的高效的访问。其实缓存机制依然存在，只是下移到 VFS文件系统层面了。

流式读

nodejs的fs模块并没有提供一个copy的方法，但我们可以很容易的实现一个，比如：

```
var source = fs.readFileSync('/path/to/source', {encoding: 'utf8'
});
fs.writeFileSync('/path/to/dest', source);
```

这种方式是把文件内容全部读入内存，然后再写入文件，对于小型的文本文件，这没有多大问题。但是对于体积较大的二进制文件，比如音频、视频文件，动辄几个GB大小，如果使用这种方法，很容易使内存“爆仓”。具体的说，对于32位系统是1GB，64位是2GB。

理想的方法应该是读一部分，写一部分，不管文件有多大，只要时间允许，总会处理完成，这里就需要用到流的概念。

上面的文件复制可以简单实现一下：

```
// pipe自动调用了data,end等事件
fs.createReadStream('/path/to/source').pipe(fs.createWriteStream(
'/path/to/dest'));
```

源文件通过管道自动流向了目标文件。

总结

- 不要迷信异步， 使用时评估同步和异步的开销，包括复杂度和性能。
- 缓存策略需要综合考虑，这离不开对系统的了解(更多的涉猎)，重复缓存只会带来没必要的开销。
- 大文件的操作，使用流式操作。

参考

- https://www.zhihu.com/question/38041375

子进程 (child_process)

child_process 是Node的一个十分重要的模块，通过它可以实现创建多进程，以利用单机的多核计算资源。虽然，Nodejs天生是单线程单进程的，但是有了child_process模块，可以在程序中直接创建子进程，并使用主进程和子进程之间实现通信。

进程通信

每个进程各自有不同的用户地址空间，任何一个进程的全局变量在另一个进程中都看不到，所以进程之间要交换数据必须通过内核， 在内核中开辟一块缓冲区，进程1把数据从用户空间拷到内核缓冲区，进程2再从内核缓冲区把数据读走，内核提供的这种机制称为进程间通信。

类型	无连接	可靠	流控制	优先级
普通PIPE	N	Y	Y	N
命名PIPE	N	Y	Y	N
消息队列	N	Y	Y	N
信号量	N	Y	Y	Y
共享存储	N	Y	Y	Y
UNIX流SOCKET	N	Y	Y	N
UNIX数据包SOCKET	Y	Y	N	N

- 注:无连接: 指无需调用某种形式的open,就有发送消息的能力流控制:

Node 中实现 IPC 通信的是管道技术，但只是抽象的称呼，具体细节实现由 libuv提供， 在 windows 下由命名管道（named pipe）实现， *nix 系统则采用 Unix Domain Socket实现。 也就是上表中的最后第二个。

Socket API原本是为网络通讯设计的，但后来在socket的框架上发展出一种IPC机制，就是UNIX Domain Socket。虽然网络socket也可用于同一台主机的进程间通讯（通过loopback地址127.0.0.1），但是UNIX Domain Socket用于IPC更有效率：不需要经过网络协议栈，不需要打包拆包、计算校验和、维护序号和应答等，只是将应用层数据从一个进程拷贝到另一个进程。

> Depending on the platform, unix domain sockets can achieve around 50% more throughput than the TCP/IP loopback (on Linux for instance).

这是因为，IPC机制本质上是可靠的通讯，而网络协议是为不可靠的通讯设计的。UNIX Domain Socket也提供面向流和面向数据包两种API接口，类似于TCP和UDP，但是面向消息的UNIX Domain Socket也是可靠的，消息既不会丢失也不会顺序错乱。

创建子进程

- spawn()启动一个子进程来执行命令
- exec()启动一个子进程来执行命令, 带回调参数获知子进程的情况, 可指定进程运行的超时时间
- execFile()启动一个子进程来执行一个可执行文件, 可指定进程运行的超时时间
- fork() 与spawn()类似, 不同在于它创建的node子进程只需指定要执行的js文件模块即可

  ```
  // don't call this example code
  var cp = require('child_process');
  cp.spawn('node', ['work.js']);
  cp.exec('node work.js', function(err, stdout, stderr) {
  // some code
  });
  cp.execFile('work.js', function(err, stdout, stderr) {
  // some code
  });
  cp.fork('./work.js');
  ```

exec方法会直接调用bash（/bin/sh程序）来解释命令，所以如果有用户输入的参数，exec方法是不安全的。

```
var path = ";user input";
child_process.exec('ls -l ' + path, function (err, data) {
  console.log(data);
});
```

上面代码表示，在bash环境下， `ls -l; user input` 会直接运行。如果用户输入恶意代码，将会带来安全风险。因此，在有用户输入的情况下，最好不使用exec方法，而是使用execFile方法。

建立 IPC 通道

父进程在创建子进程前创建IPC通道并监听, 用环境变量NODE_CHANNEL_FD告诉子进程的IPC的文件描述符。

```
startup.processChannel = function() {
  // If we were spawned with env NODE_CHANNEL_FD then load that
up and
  // start parsing data from that stream.
  if (process.env.NODE_CHANNEL_FD) {
    var fd = parseInt(process.env.NODE_CHANNEL_FD, 10);
    assert(fd >= 0);

    // Make sure it's not accidentally inherited by child proces
ses.
    delete process.env.NODE_CHANNEL_FD;

    var cp = NativeModule.require('child_process');

    // Load tcp_wrap to avoid situation where we might immediate
ly receive
    // a message.
    // FIXME is this really necessary?
    process.binding('tcp_wrap');

    cp._forkChild(fd);
    assert(process.send);
  }
};
```

子进程在启动的过程中连接IPC的FD

```
exports._forkChild = function(fd) {
  // set process.send()
  var p = new Pipe(true);
  p.open(fd);
  p.unref();
  const control = setupChannel(process, p);
  process.on('newListener', function(name) {
    if (name === 'message' || name === 'disconnect') control.ref
();
  });
  process.on('removeListener', function(name) {
    if (name === 'message' || name === 'disconnect') control.unr
ef();
  });
};
```

建立连接后父子进程就可以自由的，全双工的通信了。

句柄传递

ChildProcess 类的实例，通过调用 ChildProcess#send(message[, sendHandle[, options]][, callback]) 方法，我们可以实现与子进程的通信，其中的 sendHandle 参数支持传递 net.Server ，net.Socket 等多种句柄，使用它，我们可以很轻松的实现在进程间转发 TCP socket。

send方法可以发送的对象包括如下集中：

- net.Socket对象: TCP套接字
- net.Server对象: TCP服务器
- net.Native: C++层面的TCP套接字和IPC管道
- dgram.Socket: UDP套接字
- dgram.Native: C++层面的UDP套接字

传递的过程：

主进程 ：

- 传递消息和句柄。
- 将消息包装成内部消息，使用 JSON.stringify 序列化为字符串。

- 通过对应的 handleConversion[message.type].send 方法序列化句柄。
- 将序列化后的字符串和句柄发入 IPC channel 。

子进程

- 使用 JSON.parse 反序列化消息字符串为消息对象。
- 触发内部消息事件（internalMessage）监听器。
- 将传递来的句柄使用 handleConversion[message.type].got 方法反序列化为 JavaScript 对象。
- 带着消息对象中的具体消息内容和反序列化后的句柄对象，触发用户级别事件。

总结

很多应用比如 redis提供了本地访问的接口，进程通信使用的是 socket 的回环地址。当然它是通用性的考虑，否则要区分本地环境还是网络环境，如果不考虑这点，其实可以用 unix domain socket 代替，以获取更好的相互性能。

> Here you have the results on a single CPU 3.3GHz Linux machine :

类型	TCP	UDS	PIPE
latency	6us	2us	2us
throughput	253702 msg/s	1733874 msg/s	1682796 msg/s

- UDS: UNIX Domain Socket

参考

[1]. https://github.com/rigtorp/ipc-bench

cluster

背景

众所周知，Node.js是单线程的，一个单独的Node.js进程无法充分利用多核。Node.js从v0.8开始，新增cluster模块，让Node.js开发Web服务时，很方便的做到充分利用多核机器。

充分利用多核的思路是：使用多个进程处理业务。cluster模块封装了创建子进程、进程间通信、服务负载均衡。有两类进程，master进程和worker进程，master进程是主控进程，它负责启动worker进程，worker是子进程、干活的进程。

竞争模型

最初的 Node.js 多进程模型就是这样实现的，master 进程创建 socket，绑定到某个地址以及端口后，自身不调用 listen 来监听连接以及 accept 连接，而是将该 socket 的 fd 传递到 fork 出来的 worker 进程，worker 接收到 fd 后再调用 listen，accept 新的连接。但实际一个新到来的连接最终只能被某一个 worker 进程 accpet 再做处理，至于是哪个 worker 能够 accept 到，开发者完全无法预知以及干预。这势必就导致了当一个新连接到来时，多个 worker 进程会产生竞争，最终由胜出的 worker 获取连接。

相信到这里大家也应该知道这种多进程模型比较明显的问题了

- 多个进程之间会竞争 accpet 一个连接，产生惊群现象，效率比较低。
- 由于无法控制一个新的连接由哪个进程来处理，必然导致各 worker 进程之间的负载非常不均衡。

round-robin (轮训)

上面的多进程模型存在诸多问题，于是就出现了基于round-robin的另一种模型。 主要思路是master进程创建socket，绑定好地址以及端口后再进行监听。该socket的fd不传递到各个worker进程，当master进程获取到新的连接时，再决定将accept到的客户端socket fd传递给指定的worker处理。我这里使用了指定, 所以如何传递以及传递给哪个worker完全是可控的，round-robin只是其中的某种算法而已，当然可以换成其他的。

Master是如何将接收的请求传递至worker中进行处理然后响应的？

Cluster 模块通过监听该内部TCP服务器的connection事件，在监听器函数里，有负载均衡地挑选出一个worker，向其发送newconn内部消息（消息体对象中包含cmd: 'NODE_CLUSTER'属性）以及一个客户端句柄（即connection事件处理函数的第二个参数），相关代码如下：

```
// lib/cluster.js
// ...

function RoundRobinHandle(key, address, port, addressType, backlog, fd) {
  // ...
  this.server = net.createServer(assert.fail);
  // ...

  var self = this;
  this.server.once('listening', function() {
    // ...
    self.handle.onconnection = self.distribute.bind(self);
  });
}

RoundRobinHandle.prototype.distribute = function(err, handle) {
  this.handles.push(handle);
  var worker = this.free.shift();
  if (worker) this.handoff(worker);
};

RoundRobinHandle.prototype.handoff = function(worker) {
  // ...
  var message = { act: 'newconn', key: this.key };
  var self = this;
  sendHelper(worker.process, message, handle, function(reply) {
    // ...
  });
};
```

Worker进程在接收到了newconn内部消息后，根据传递过来的句柄，调用实际的业务逻辑处理并返回：

```
// lib/cluster.js
// ...

// 该方法会在Node.js初始化时由 src/node.js 调用
cluster._setupWorker = function() {
  // ...
  process.on('internalMessage', internal(worker, onmessage));

  // ...
  function onmessage(message, handle) {
    if (message.act === 'newconn')
      onconnection(message, handle);
    // ...
  }
};

function onconnection(message, handle) {
  // ...
  var accepted = server !== undefined;
  // ...
  if (accepted) server.onconnection(0, handle);
}
```

至此，也总结一下：

- 所有请求先同一经过内部TCP服务器。
- 在内部TCP服务器的请求处理逻辑中，有负载均衡地挑选出一个worker进程，将其发送一个newconn内部消息，随消息发送客户端句柄。
- Worker进程接收到此内部消息，根据客户端句柄创建net.Socket实例，执行具体业务逻辑，返回。

listen 端口复用

为了得到这个问题的解答，我们先从worker进程的初始化看起，master进程在fork工作进程时，会为其附上环境变量NODE_UNIQUE_ID，是一个从零开始的递增数：

```
// lib/cluster.js
// ...

function createWorkerProcess(id, env) {
  // ...
  workerEnv.NODE_UNIQUE_ID = '' + id;

  // ...
  return fork(cluster.settings.exec, cluster.settings.args, {
    env: workerEnv,
    silent: cluster.settings.silent,
    execArgv: execArgv,
    gid: cluster.settings.gid,
    uid: cluster.settings.uid
  });
}
```

随后Node.js在初始化时，会根据该环境变量，来判断该进程是否为cluster模块fork出的工作进程，若是，则执行workerInit()函数来初始化环境，否则执行masterInit()函数。

在workerInit()函数中，定义了cluster._getServer方法，这个方法在任何net.Server实例的listen方法中，会被调用：

```
// lib/net.js
// ...

function listen(self, address, port, addressType, backlog, fd, e
xclusive) {
  exclusive = !!exclusive;

  if (!cluster) cluster = require('cluster');

  if (cluster.isMaster || exclusive) {
    self._listen2(address, port, addressType, backlog, fd);
    return;
  }

  cluster._getServer(self, {
    address: address,
    port: port,
    addressType: addressType,
    fd: fd,
    flags: 0
  }, cb);

  function cb(err, handle) {
    // ...

    self._handle = handle;
    self._listen2(address, port, addressType, backlog, fd);
  }
}
```

你可能已经猜到，答案就在这个cluster._getServer函数的代码中。它主要干了两件事：

- 向master进程注册该worker，若master进程是第一次接收到监听此端口/描述符下的worker，则起一个内部TCP服务器，来承担监听该端口/描述符的职责，随后在master中记录下该worker。
- Hack掉worker进程中的net.Server实例的listen方法里监听端口/描述符的部分，使其不再承担该职责。

对于第一件事，由于master在接收，传递请求给worker时，会符合一定的负载均衡规则（在非Windows平台下默认为轮询），这些逻辑被封装在RoundRobinHandle类中。故，初始化内部TCP服务器等操作也在此处：

```
// lib/cluster.js
// ...

function RoundRobinHandle(key, address, port, addressType, backl
og, fd) {
  // ...
  this.handles = [];
  this.handle = null;
  this.server = net.createServer(assert.fail);

  if (fd >= 0)
    this.server.listen({ fd: fd });
  else if (port >= 0)
    this.server.listen(port, address);
  else
    this.server.listen(address);  // UNIX socket path.

  /// ...
}
```

对于第二件事，由于net.Server实例的listen方法，最终会调用自身_handle属性下listen方法来完成监听动作，故在代码中修改之：

```
// lib/cluster.js
// ...

function rr(message, cb) {
  // ...
  // 此处的listen函数不再做任何监听动作
  function listen(backlog) {
    return 0;
  }

  function close() {
    // ...
```

```
  }
  function ref() {}
  function unref() {}

  var handle = {
    close: close,
    listen: listen,
    ref: ref,
    unref: unref,
  };
  // ...
  handles[key] = handle;
  cb(0, handle); // 传入这个cb中的handle将会被赋值给net.Server实例中
的_handle属性
}

// lib/net.js
// ...
function listen(self, address, port, addressType, backlog, fd, e
xclusive) {
  // ...

  if (cluster.isMaster || exclusive) {
    self._listen2(address, port, addressType, backlog, fd);
    return; // 仅在worker环境下改变
  }

  cluster._getServer(self, {
    address: address,
    port: port,
    addressType: addressType,
    fd: fd,
    flags: 0
  }, cb);

  function cb(err, handle) {
    // ...
    self._handle = handle;
    // ...
  }
```

```
}
```

至此，总结下：

- 端口仅由master进程中的内部TCP服务器监听了一次。
- 不会出现端口被重复监听报错，是由于，worker进程中，最后执行监听端口操作的方法，已被cluster模块主动覆盖。

总结

参考

javascript 的坑

弱类型

- 判断对象是否不存在使用if(!obj), 此时如果obj为'' 或 0 或 false,也会误认为不存在; 正确的写法也是纷繁复杂：见参考
- 字符串连接,一般会自动转成string, 但不巧如果头两个正好可以转数字,那么它会按数字相加.如:

```
var a = 1,b = 'b',c = 10;
var s1 = a + c + b;
var s2 = '' + a + c + b;
console.log(s1); //"11b"
console.log(s2); //"110b"
```

保险起见：注意使用 `Number` 转换。

this指针

- 一般调用时，this是窗口的全局对象，比如在浏览器中就是window对象
- `.call` 和 `.apply` 方法调用时可以改变 `this` 的值
- 在 `prototype` 函数内部，this指向该类创造的实例对象

排序 sort

JavaScript的Array的sort()方法就是用于排序的，但是排序结果可能让你大吃一惊：

```
[10, 20, 1, 2].sort(); // [1, 10, 2, 20]
```

这是因为Array的sort()方法默认把所有元素先转换为String再排序，结果'10'排在了'2'的前面，因为字符'1'比字符'2'的ASCII码小。

如果不知道sort()方法的默认排序规则，直接对数字排序，绝对栽进坑里！ 幸运的是，sort()方法也是一个高阶函数，它还可以接收一个比较函数来实现自定义的排序。

异步处理

- 错误处理忘记 `return` .
- 一个异步方法中处理不当有可能会多次触发callback,又或者是一个callback都没触发,需要仔细处理逻辑流程.

科学计算

- v8 引擎中 js 的 Number 对象的内部实现只有两种，一是smi（也就是小整数），二是 `double` 。Node.js 根本没有 float! 如果使用了 float , 注意存在精度的缺失。
- 位运算， javascript 只支持32位，超过32位的，需要用大数模拟。 https://github.com/justmoon/node-bignum

参考

- http://www.ruanyifeng.com/blog/2011/05/how_to_judge_the_existence_of_a_global_object_in_javascript.html

Node.js在物联网

1、Build Node.js for Android

Linux构建环境

```
jiangling@young:~/node/deps/npm$ uname -a
Linux young 3.11.0-15-generic #25~precise1-Ubuntu SMP Thu Jan 30
 17:39:31 UTC 2014 x86_64 x86_64 x86_64 GNU/Linux
```

32位 Android NDK

```
drwxr-xr-x 10 jiangling jiangling      4096  3月   1  2014 androi
d-ndk-r9d
```

clone node.js

```
git clone https://github.com/joyent/node.git
```

android-configure patch

```
jiangling@young:~/node$ git diff
diff --git a/android-configure b/android-configure
index 7acb7f3..aae0bf1 100755
--- a/android-configure
+++ b/android-configure
@@ -3,7 +3,7 @@
 export TOOLCHAIN=$PWD/android-toolchain
 mkdir -p $TOOLCHAIN
 $1/build/tools/make-standalone-toolchain.sh \
-    --toolchain=arm-linux-androideabi-4.7 \
+    --toolchain=arm-linux-androideabi-4.8 \
     --arch=arm \
     --install-dir=$TOOLCHAIN \
     --platform=android-9
```

否则会出现 `arm-linux-androideabi-gcc not found` 的错误。

configure && make

```
source ./android-configure ~/android-ndk-r9b
mv python2.7 oldpython2.7
ln -s /usr/bin/python2.7 python2.7
cd  ~/node
make
```

node bin大小

```
jiangling@young:~/node$ file out/Release/node
out/Release/node: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), not stripped

root@android:/data/local/tmp # ls -al node
ls -al node
-rwx------ shell    shell    12158228 2014-11-24 17:01 node
```

配置without-ssl后大小

```
jiangling@young:~/node$ ls -al out/Release/node
-rwxrwxr-x 1 jiangling jiangling 9804644 11月 26 13:55 out/Relea
se/node
```

2、Run Node.js on Android

这边我选择了ROOT过的小米M1手机，安装了"瑞士军刀"busybox, 以便查看系统信息。

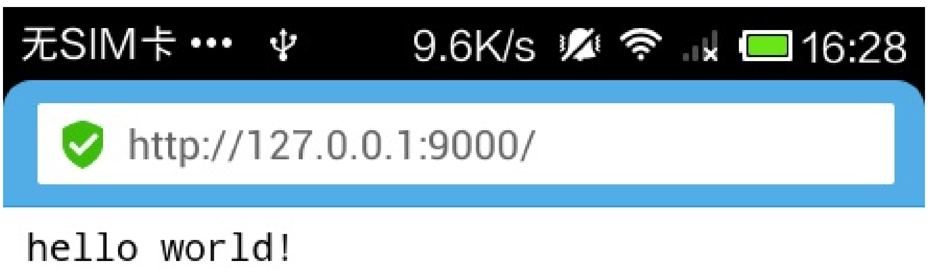
```
root@android:/data/local/tmp # ./busybox uname -a
./busybox uname -a
Linux localhost 3.4.0-perf-g1ccebb5-00146-gd6845ec #1 SMP PREEMP
T Mon Nov 4 20:10:00 CST 2013 armv7l GNU/Linux
```

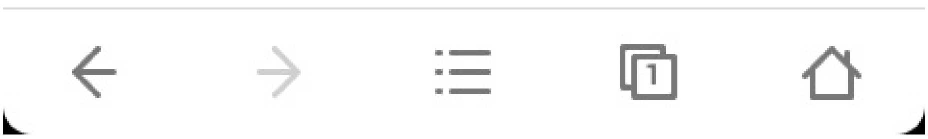
用ADB将 `node` , `test.js` (经典的hello world) push到M1上：

```
adb push d:\node /data/local/tmp
adb push d:\test.js /data/local/tmp
adb shell chmod 755 /data/local/tmp/node
```


无SIM卡 10:38
窗口 1
sion denied
7 busybox
root@android:/data/local/tmp # ./busybox uname -a
Linux localhost 3.4.0-perf-g1ccebb5-00146-gd6845ec #1 SMP PREEMPT Mon Nov 4 20:10:00 CST 2013 armv7l GNU/Linux
node t.js
Server running at http://127.0.0.1:9000/ , pid-> 8543
123
符
中/英

运行和结果


无SIM卡 9.6K/s 16:28
http://127.0.0.1:9000/
hello world!



Node.js Docker

docker vs host

```
var http = require('http');
    http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World\n');
}).listen(1337);
console.log('Server running at http://0.0.0.0:1337/');
```

针对我们最关心的性能问题，用 `node-v4.2.3` 做了对比测试： `docker` vs `host` node 。结论是性能损失在1%~4%之间，依据网络，业务代码因子而不定，数据如下：

- 外部网络环境

```
docker node
root@ubuntu-512mb-nyc3-01:~/wrk# ./wrk http://192.241.209.*:1337
Running 10s test @ http://192.241.209.*:1337
  2 threads and 10 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    76.01ms    1.24ms  88.79ms   83.68%
    Req/Sec    65.51     16.12   101.00     76.77%
  1305 requests in 10.04s, 198.81KB read
Requests/sec:    129.99
Transfer/sec:     19.80KB
host node
root@ubuntu-512mb-nyc3-01:~/wrk# ./wrk http://192.241.209.*:1337

Running 10s test @ http://192.241.209.*:1337
  2 threads and 10 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    75.85ms    1.87ms  87.10ms   61.29%
    Req/Sec    65.66     12.35   101.00     57.07%
  1307 requests in 10.03s, 199.11KB read
Requests/sec:    130.27
Transfer/sec:     19.85KB
```

- 内部网络环境

```
** host node**
[root@centos7-x64 statusbar]# wrk  http://localhost:1337
Running 10s test @ http://localhost:1337
  2 threads and 10 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     1.51ms    1.34ms  39.81ms   98.03%
    Req/Sec     3.46k   821.79     4.29k    76.50%
  68971 requests in 10.05s, 10.26MB read
Requests/sec:   6862.84
Transfer/sec:      1.02MB

docker node `--net=host`模式
[root@centos7-x64 ~]# wrk http://localhost:1337
Running 10s test @ http://127.0.0.1:1337
  2 threads and 10 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     1.49ms  287.14us   3.81ms   92.99%
    Req/Sec     3.30k   424.64     3.60k    91.00%
  65866 requests in 10.03s, 9.80MB read
Requests/sec:   6564.82
Transfer/sec:      0.98MB
```

总的来说，对于正常 web 应用，走外部网络连接，性能损失很小，却极大的方便了我们的开发运维。

部署应用

> 什么是 MEAN 架构？ MEAN 表示 Mongodb / ExpressJS / AngularJS /NodeJS，是目前流行的网站应用开发组合，涵盖前端至后台。由于这些框架用的语言都是 Javascript，所以又戏称 Javascript Fullstack。

本例子中，我们将尝试部署一个 MEAN 架构的 NodeJS 应用。

```
目录结构
├── docker-node-full/
|   ├── start.sh
|   ├── Dockerfile
```

安装 Docker

Ubuntu的系统，使用 apt 安装：

```
$ sudo apt-get install -y docker-engine
```

创建步骤

- 创建文件夹

  ```
  mkdir ~/docker-node-full && cd $_
  ```

- 创建 Dockerfile 配置文件

```
# 设置基础镜像
FROM ubuntu:14.10

# 安装 NodeJS 和 npm
RUN apt-get install -y nodejs npm

# 由于 apt-get 下载的 Node 实际上是 nodejs，所以要创建一个 node 的快捷
方式
RUN ln -s /usr/bin/nodejs /usr/bin/node

# 安装 Git
RUN apt-get install -y git

# 安装 Mongodb（来自官方教程）
RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv
 7F0CEB10
RUN echo 'deb http://downloads-distro.mongodb.org/repo/ubuntu-up
start dist 10gen' | tee /etc/apt/sources.list.d/mongodb.list
RUN apt-get update
RUN apt-get install -y mongodb-org

# 设置工作目录
WORKDIR /srv/full
```

```
# 清空已存在的文件（如果有）
RUN rm -rf /srv/full

# 通过 Git 下载准备好的 MEAN 架构的网站代码
RUN git clone https://github.com/chuyik/fullstack-demo-dist.git
.

# 安装 NodeJS 依赖库
RUN npm install --production

# 创建 mongodb 数据文件夹
RUN mkdir -p /data/db

# 暴露端口（分别是 NodeJS 应用和 Mongodb）
EXPOSE 5566 27017

# 设置 NodeJS 应用环境变量
ENV NODE_ENV=production PORT=5566

# 添加启动脚本
ADD start.sh /tmp/
RUN chmod +x /tmp/start.sh

# 设置启动时默认运行命令
CMD ["bash", "/tmp/start.sh"]
```

- 创建 start.sh 启动脚本

```
# 后台启动 Mongodb
mongod --fork --logpath=/var/log/mongo.log --logappend

# 运行 NodeJS 应用
npm start
```

构建镜像

```
# 通过该命令，按照 Dockerfile 所配置的信息构建出镜像
  docker build --rm -t node-full .

# 检查镜像是否创建成功
  docker images
```

运行镜像

```
# 运行刚刚创建的镜像
# -p 设置端口，格式为「主机端口:容器端口」
  docker run -p 5566:5566 node-full
```

访问应用

可以用浏览器访问 http://localhost:5566, 或运行 curl -s http://localhost:5566。

保存 **Mongodb** 数据文件

由于 Mongodb 服务运行在 Docker 容器 (container) 中，所以数据也在里面，但这并不利于数据管理和保存。因此，可以通过一些方法，将 Mongodb 数据文件保存在容器的外头。

磁盘映射

这个是最简单的方式，在 docker run 命令当中，就有磁盘映射的参数 -v。

```
# -v 磁盘映射，格式为「主机目录:容器目录」
docker run -p 5566:5566 -v /var/mongodata:/data/db node-full
```

但这个命令在 Mac 和 Windows 中执行失败，因为 boot2docker 的虚拟机不支持。所以，可以将数据保存在 boot2docker 内，并设置共享文件夹便于 Mac 或 Windows 访问。

楔子

`node-profiler` 作为 alinode 团队的另一款产品能够帮助您线下深入分析 javascript 代码的性能，将Google V8的性能细节展现在您的面前，优化而知其所以然。线上调优请使用 `alinode` 。

下载安装

推荐安装工具 `tnvm` ，支持 node, alinode, profiler 的安装切换。

```
wget -O- https://raw.githubusercontent.com/aliyun-node/tnvm/mast
er/install.sh | bash
```

完成安装后，您需要将tnvm添加为命令行程序。根据平台的不同，可能是 ~/.bashrc，~/.profile或~/.zshrc.

```
 tnvm install profiler-v0.12.6
 tnvm use profiler-v0.12.6
```

使用示例

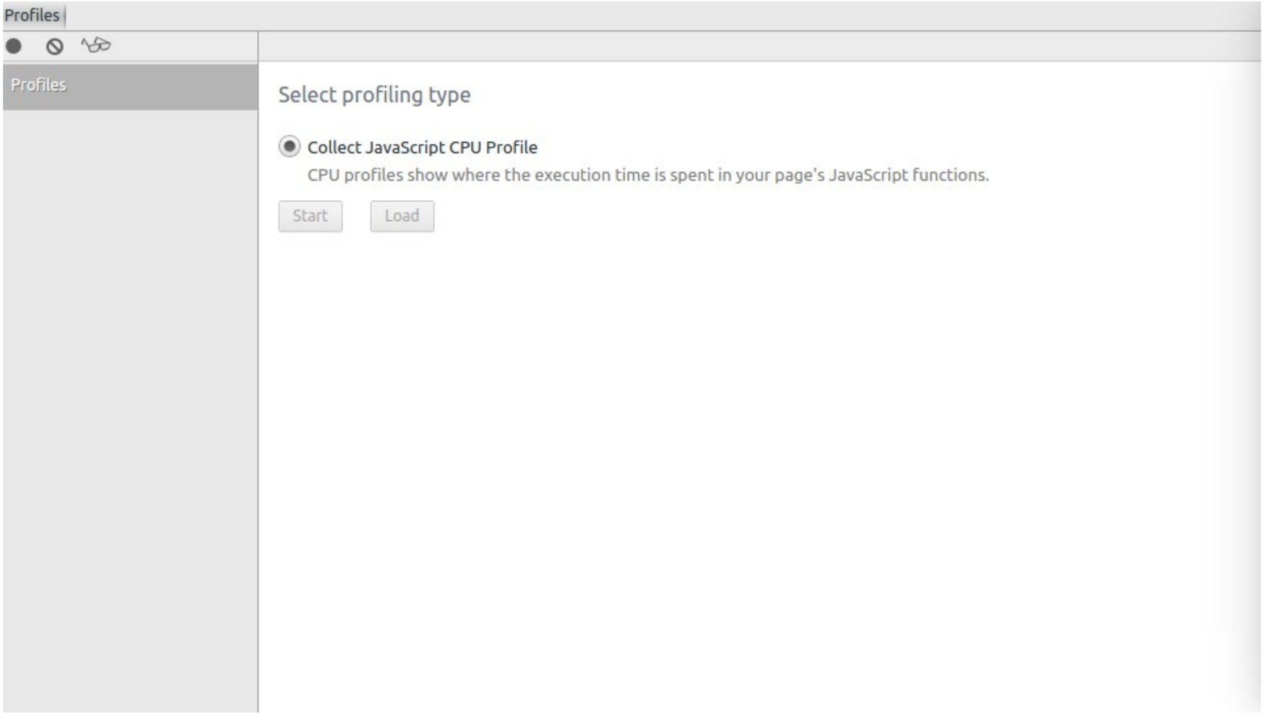
```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200);
  res.end('hello world!');
}).listen(1334);
```

```
$ node-profiler server.js
start agent
webkit-devtools-agent: A proxy got connected.
webkit-devtools-agent: Waiting for commands...
webkit-devtools-agent: Websockets service started on 0.0.0.0:999
9  <==启动成功
```

如出现如下：

```
Error: listen EADDRINUSE          <== 可能是由于端口被占用
```

成功启动后，则用chrome(推荐)手动打开url (http://alinode.aliyun.com/profiler/inspector.html?host=localhost:9999&page=0) 出现如下界面：

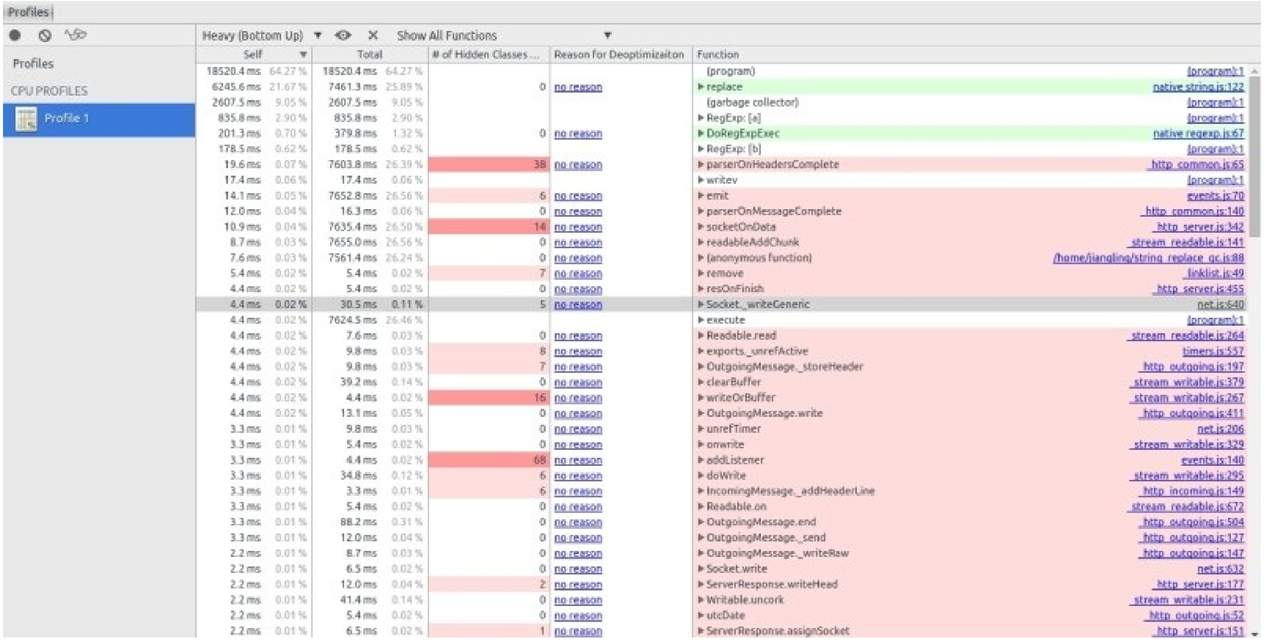


默认**Collect JavaSript CPU Profile**，单击**Start**。

可以采用压测脚本实现对服务进行压力测试，保证更多的结果：

```
$ wrk http://localhost:1334/ # 这里使用wrk，也可以使用其他工具，如ab
```

点击**Stop**，得到如下图的结果：



可以看到更多关于函数在运行时的信息。

UI含义

UI栏目	示意
Self	exclusive time
Total	inclusive time
# Hidden Classes	隐藏类个数
Bailout	v8中提取的最后一次去优化原因
Function	函数名称 script : line

红色表示函数未被优化， 淡绿色表示函数被**V8**优化过。

原理介绍

- 基于V8内置采样收集器；
- 固定采样频率，默认1ms, 可配置；
- 会暂停主线程，采样函数call stack，统计时间；
- 需保证采样足够长的时间（预热）。

解释下两个概念

- exclusive time :独占时间
- inclusive time :包含时间

```
function foo() {
    bar();
}
function bar() {
          <==采样点
}
foo();
```

这个例子，采样点在bar()，那么bar所消耗的时间叫作 **exclusive time**，而foo调用了bar, foo所消耗的时间包括了bar的时间，叫作 **inclusive time** .

注意事项

- 该工具目前只支持 X64 平台（Linux, Mac)。
- 切勿部署到线上，如需线上调优请使用 alinode。

V8 bailout reasons

v8 bailout reasons 的例子, 解释和建议. 帮助 `alinode` 的用户根据 CPU-Profiler 的提示进行优化。

索引

Bailout reasons

Bailout reasons

Assignment to parameter in arguments object

- 简单例子

```
// sloppy mode only
function test(a) {
  if (arguments.length < 2) {
    a = 0;
  }
}
```

- Why
    - 只会在函数中重新赋值参数发生。
- Advices
    - 你不能给变量 a 重新赋值.
    - 最好使用 strict mode .
    - V8 最新的 TurboFan 会有优化 #1.

Bad value context for arguments value

- 简单例子

```
// strict & sloppy modes
function test1() {
  arguments[0] = 0;
}

// strict & sloppy modes
function test2() {
  arguments.length = 0;
}

// strict & sloppy modes
function test3() {
  return arguments;
}

// strict & sloppy modes
function test4() {
  var args = [].slice.call(arguments);
}

// strict & sloppy modes
function test5() {
  var a = arguments;
  return function() {
    return a;
  };
}
```

- Why
  - 要求再具体化 `arguments` 数组.
- Advices
  - 可以读读: https://github.com/petkaantonov/bluebird/wiki/Optimization-killers#3-managing-arguments
  - 你可以循环 `arguments` 创建一个新的数组 Unsupported phi use of arguments
  - V8 最新的 TurboFan 会有优化 #1.
- 外部链接

  - https://github.com/bevry/taskgroup/issues/12
  - 更多

ForInStatement with non-local each variable

- 简单例子

```
// strict & sloppy modes
function test1() {
  var obj = {};
  for(key in obj);
}

// strict & sloppy modes
function key() {
  return 'a';
}
function test2() {
  var obj = {};
  for(key() in obj);
}
```

- Why
  - https://github.com/yjhjstz/v8-git-mirror/blob/master/src/hydrogen.cc#L5254
- Advices
  - 只有纯局部变量可以用于 for...in
  - https://github.com/petkaantonov/bluebird/wiki/Optimization-killers#5-for-in
- 外面链接
  - https://github.com/mbostock/d3/pull/2686

Object literal with complex property

- 简单例子

```
// strict & sloppy modes
function test() {
  return {
    __proto__: 3
  };
}
```

- Why
- Advices
  - 简化 Object。

ForInStatement is not fast case

- 简单例子

  ```
  for (var prop in obj) {
  /* lots of code */
  }
  ```

- Why
  - for 循环中包含太多的代码。
- Advices
  - for 循环中的提取代码提取为函数。

Reference to a variable which requires dynamic lookup

- 简单例子

```
// sloppy mode only
function test() {
with ({x:1}) {
  return x;
}
}
```

- Why
    - 编译时编译定位失败，Crankshaft需要重新动态查找。#3
- Advices
    - TurboFan可以优化。

TryCatchStatement

- 简单例子

```
// strict & sloppy modes OR // sloppy mode only
function func() {
  return 3;
  try {} catch(e) {}
}
```

- Why
    - try/catch 使得控制流不稳定，很难在运行时优化。
- Advices
    - 不要在负载重的函数中使用try/catch.
    - 可以重构为 `try { func() } catch`

TryFinallyStatement

- 简单例子

```
// strict & sloppy modes OR // sloppy mode only
function func() {
  return 3;
  try {} finally {}
}
```

- Why
  - See TryCatchStatement
- Advices
  - See TryCatchStatement

Unsupported phi use of arguments

- 简单例子

```
// strict & sloppy modes
function test1() {
  var _arguments = arguments;
  if (0 === 0) { // anything evaluating to true, except a number
 or `true`
    _arguments = [0]; // Unsupported phi use of arguments
  }
}

// strict & sloppy modes
function test2() {
  var _arguments = arguments;
  for (var i = 0; i < 1; i++) {
    _arguments = [0]; // Unsupported phi use of arguments
  }
}

// strict & sloppy modes
function test3() {
  var _arguments = arguments;
  var again = true;
  while (again) {
    _arguments = [0]; // Unsupported phi use of arguments
    again = false;
  }
}
```

- Why
  - Crankshaft 无法知道 `_arguments` 是 object 或 array.
  - 深入了解
- Advices
  - 最好操作 `arguments` 的拷贝.
  - TurboFan 可以优化 #1.

Yield

- 简单例子

```
// strict & sloppy modes
function* test() {
  yield 0;
}
```

- Why
    - generator 状态保持、恢复通过拷贝函数栈帧实现，但在优化编译器中并不适用。
- Advices
    - 暂时不用考虑，TurboFan 可以优化。
- 外部链接：
    - https://groups.google.com/forum/#!topic/v8-users/KnnUb-u4rA8

Resources

- All bailout reasons in Chromium codebase
- Bad value context for arguments value
- I-want-to-optimize-my-JS-application-on-V8 checklist
- JavaScript: Performance loss on incorrect arguments using
- Optimization killers
- OptimizationKillers
- Performance Tips for JavaScript in V8
- thlorenz/v8-perf